The SGRUD Thesis

SGRUD is Growing Rapidly Until Distinction.

Philip Schildkamp

Abstract

Abstract.

Contents

List of Figures

List of Tables

Preformatted

```
*Text*
```

Table

row	row	row
col	col	col
col	col	col
col	col	col

References

Appendix

Module: bin

bin

• **bin**: `Object`

`@sgrud/bin` - The SGRUD CLI.

```
Description
  @sgrud/bin - The SGRUD CLI

Usage
  $ sgrud <command> [options]

Available Commands
  construct    Builds a SGRUD-based project using `microbundle`
  kickstart    Kickstarts a SGRUD-based project using `simple-git`
  postbuild    Replicates exported package metadata for SGRUD-based projects
  runtimify    Creates ESM or UMD bundles for ES6 modules using `microbundle`
  universal    Runs SGRUD in universal (SSR) mode using `puppeteer`

For more info, run any command with the `--help` flag
  $ sgrud construct --help
  $ sgrud kickstart --help

Options
  -v, --version    Displays current version
  -h, --help       Displays this message
```

Defined in

packages/bin/index.ts:34

---

bin.construct

construct

▸ **construct**(`options?`): `Promise<void>`

Constructs a SGRUD-based project using microbundle.

```
Description
  Constructs a SGRUD-based project using `microbundle`

Usage
  $ sgrud construct [...modules] [options]

Options
  --compress    Compress/minify build output  (default true)
  --format      Build specified formats  (default commonjs,modern,umd)
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud construct # Run with default options
```

```
  $ sgrud construct ./project/module # Build ./project/module
  $ sgrud construct ./module --format umd # Build ./module as umd
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.construct();
```

Example

Construct `./project/module`:

```
require('@sgrud/bin');

sgrud.bin.construct({
  modules: ['./project/module']
});
```

Example

Construct `./module` as `umd`:

```
require('@sgrud/bin');

sgrud.bin.construct({
  modules: ['./module'],
  format: 'umd'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.compress?`	`boolean`	Compress/minify **construct** output. **`Default Value`** `true`
`options.format?`	`string`	**Construct** specified formats. **`Default Value`** `'commonjs,modern,umd'`
`options.modules?`	`string[]`	Modules to **construct**. **`Default Value`** `package.json#sgrud.construct`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/construct.ts:77

---

bin.kickstart

kickstart

▸ **kickstart**(`options?`): `Promise<void>`

Kickstarts a SGRUD-based project using simple-git.

```
Description
  Kickstarts a SGRUD-based project using `simple-git`

Usage
  $ sgrud kickstart [library] [options]

Options
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message
```

```
Examples
  $ sgrud kickstart # Run with default options
  $ sgrud kickstart preact --prefix ./module # Kickstart preact in ./module
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.kickstart();
```

Example

Kickstart `preact` in `./module`:

```
require('@sgrud/bin');

sgrud.bin.kickstart({
  prefix: './module',
  library: 'preact'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.library?`	`string`	Library which to base upon. **`Default Value`** `'sgrud'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/kickstart.ts:59

---

bin.postbuild

postbuild

▸ **postbuild**(`options?`): `Promise<void>`

Replicates exported package metadata for SGRUD-based projects.

```
Description
  Replicates exported package metadata for SGRUD-based projects

Usage
  $ sgrud postbuild [...modules] [options]

Options
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud postbuild # Run with default options
  $ sgrud postbuild ./project/module # Postbuild ./project/module
  $ sgrud postbuild --prefix ./module # Run in ./module
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.postbuild();
```

Example

Postbuild `./project/module`:

```
require('@sgrud/bin');

sgrud.bin.postbuild({
  modules: ['./project/module']
});
```

Example

Run in `./module`:

```
require('@sgrud/bin');

sgrud.bin.postbuild({
  prefix: './module'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.modules?`	`string[]`	Modules to **postbuild**. **`Default Value`** `package.json#sgrud.postbuild`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/postbuild.ts:70

---

bin.runtimify

runtimify

▸ **runtimify**(`options?`): `Promise<void>`

Creates ESM or UMD bundles for node modules using microbundle.

```
Description
  Creates ESM or UMD bundles for node modules using `microbundle`

Usage
  $ sgrud runtimify [...modules] [options]

Options
  --format      Runtimify bundle format (umd or esm)  (default umd)
  --output      Output file in module root  (default runtimify.[format].js)
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud runtimify # Run with default options
  $ sgrud runtimify @microsoft/fast # Runtimify `@microsoft/fast`
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.runtimify();
```

Example

Runtimify `@microsoft/fast`:

```
require('@sgrud/bin');

sgrud.bin.runtimify({
  modules: ['@microsoft/fast']
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.format?`	`string`	**Runtimify** bundle format (umd or esm). **`Default Value`** `'umd'`
`options.modules?`	`string[]`	Modules to **runtimify**. **`Default Value`** `package.json#sgrud.runtimify`
`options.output?`	`string`	Output file in module root. **`Default Value`** `'runtimify.[format].js'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/runtimify.ts:63

---

bin.universal

universal

▸ **universal**(`options?`): `Promise<void>`

Runs SGRUD in **universal** (SSR) mode using puppeteer.

```
Description
  Runs SGRUD in universal (SSR) mode using `puppeteer`

Usage
  $ sgrud universal [entry] [options]

Options
  --chrome      Chrome executable path  (default /usr/bin/chromium-browser)
  --prefix      Use an alternative working directory  (default ./)
  -H, --host    Host/IP to bind to  (default 127.0.0.1)
  -p, --port    Port to bind to  (default 4000)
  -h, --help    Displays this message

Examples
  $ sgrud universal # Run with default options
  $ sgrud universal --host 0.0.0.0 # Listen on all IPs
  $ sgrud universal -H 192.168.0.10 -p 4040 # Listen on 192.168.0.10:4040
```

`Example`

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.universal();
```

`Example`

Listen on all IPs:

```
require('@sgrud/bin');

sgrud.bin.universal({
  host: '0.0.0.0'
});
```

Example

Listen on `192.168.0.10:4040`:

```
require('@sgrud/bin');

sgrud.bin.universal({
  host: '192.168.0.10',
  port: '4040'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.chrome?`	`string`	Chrome executable path. **`Default Value`** `'/usr/bin/chromium-browser'`
`options.entry?`	`string`	HTML document (relative to `prefix`). **`Default Value`** `'index.html'`
`options.host?`	`string`	Host/IP to bind to. **`Default Value`** `'127.0.0.1'`
`options.port?`	`string`	Port to bind to. **`Default Value`** `'4000'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/universal.ts:76

Module: bus

bus

• **bus**: `Object`

`@sgrud/bus` - The SGRUD Software Bus.

The functions and classes found within this module are intended to ease the internal communication of applications building upon the SGRUD client libraries. By establishing busses between different modules of an application or between an application and plugins extending it, loose coupling of data transferral and functionality can be achieved. This module includes a standalone JavaScript bundle which will be used to instantiate a Worker, which is used as central hub for data exchange.

Defined in packages/bus/index.ts:18

---

bus.BusHandle

BusHandle

T **BusHandle**: ‘${string}.${string}.${string}‘

The **BusHandle** is a string literal helper type which enforces any assigned value to contain at least three dots. It represents a type constraint which should be thought of as domain name in reverse notation. All **BusHandle**s thereby designate a hierarchical structure, which the BusHandler in conjunction with the BusWorker operate upon.

Example

Library-wide **BusHandle**:

```ts
import { BusHandle } from '@sgrud/bus';

const busHandle: BusHandle = 'io.github.sgrud';
```

Example

An invalid **BusHandle**:

```ts
import { BusHandle } from '@sgrud/bus';

const busHandle: BusHandle = 'org.example';
// Type [...] is not assignable to type 'BusHandle'.
```

See

BusHandler

Defined in packages/bus/src/handler/handler.ts:35

---

bus.BusHandler

BusHandler

• **BusHandler**: `Object`

The **BusHandler** is a Singleton class, implementing and orchestrating the establishment, transferral and deconstruction of busses in conjunction with the BusWorker process. To designate different busses, the string literal helper type BusHandle is employed. As an example, let the following hierarchical structure be given:

```
io.github.sgrud
├── io.github.sgrud.core
│   ├── io.github.sgrud.core.httpState
│   └── io.github.sgrud.core.kernel
├── io.github.sgrud.data
│   ├── io.github.sgrud.data.model.current
│   └── io.github.sgrud.data.model.global
└── io.github.sgrud.shell
    └── io.github.sgrud.shell.route
```

Depending on the BusHandle, one may subscribe to all established busses beneath the root `io.github.sgrud` handle or only to a specific bus, e.g., `io.github.sgrud.core.kernel`. The resulting Observable will either emit all values passed through all busses with their corresponding BusHandles, or only the specific scoped values, corresponding to the BusHandle.

Decorator

Singleton

See

BusWorker

Defined in packages/bus/src/handler/handler.ts:120

---

bus.BusHandler.constructor

constructor

• **new BusHandler**(`tuples?`)

Public **constructor**. As this class is a transparent Singleton, calling the `new` operator on it will always yield the same instance. The `new` operator can therefore be used to bulk-publish busses.

Example

Set the `'io.github.sgrud.example'` bus:

```ts
import { BusHandler } from '@sgrud/bus';
import { of } from 'rxjs';

new BusHandler([
```

```
  ['io.github.sgrud.example', of('published')]
]);
```

Parameters

Name	Type	Description
`tuples?`	[`${string}.${string}.${string}`, `Observable<any>`][]	List of busses to publish.

Defined in packages/bus/src/handler/handler.ts:158

---

bus.BusHandler.get

get

▸ **get**<T>(handle): `Observable<BusValue<T>>`

Invoking this method **get**s the Observable bus represented by the supplied `handle`. The method will return an Observable originating from the BusWorker which emits all BusValues published under the supplied `handle`. When **get**ting `'io.github.sgrud'`, all busses hierarchically beneath this `handle`, e.g., `'io.github.bus.status'`, will also be emitted by the returned Observable.

`Example`

Get the `'io.github.sgrud'` bus:

```
import { BusHandler } from '@sgrud/bus';

const busHandler = new BusHandler();
busHandler.get('io.github.sgrud.example').subscribe(console.log);
```

Type parameters

Name	Description
T	Bus type.

Parameters

Name	Type	Description
`handle`	`${string}.${string}.${string}`	BusHandle to **get**.

Returns `Observable<BusValue<T>>`

Observable bus for `handle`.

Defined in packages/bus/src/handler/handler.ts:192

---

bus.BusHandler.set

set

▸ **set**<T>(handle, bus): `void`

Publishes the supplied Observable `bus` under the supplied `handle`. Calling this method registers the supplied Observable with the BusWorker. When the Observable completes, the registration will self-destruct. When overwriting a registration by supplying a previously used `handle` in conjunction with a different Observable `bus`, the previously supplied Observable will be unsubscribed.

`Example`

Set the `'io.github.sgrud.example'` bus:

```ts
import { BusHandler } from '@sgrud/bus';
import { of } from 'rxjs';

const busHandler = new BusHandler();
busHandler.set('io.github.sgrud.example', of('published'));
```

Type parameters

Name	Description
T	Bus type.

Parameters

Name	Type	Description
handle	`${string}.${string}.${string}`	BusHandle to **set**.
bus	Observable<T>	Observable bus for handle.

Returns `void`

Defined in packages/bus/src/handler/handler.ts:225

---

bus.BusHandler.worker

worker

• Readonly **worker**: Thread<BusWorker>

Spawned **worker** process and main bus workhorse. The underlying BusWorker is run inside a Worker context and handles all published and subscribed busses and the aggregation of their values depending on their BusHandle, i.e., hierarchy.

Decorator

Spawn

Defined in packages/bus/src/handler/handler.ts:136

---

bus.BusValue

BusValue

• **BusValue**<T>: Object

The **BusValue** is an interface describing the shape of all values emitted by any bus. As busses are Observable streams, which are dynamically merged through their hierarchical structure and therefore may emit more than one value from more than one handle, each value emitted by any bus contains its originating handle and its typed internal value.

Example

Logging emitted **BusValue**s.

```ts
import { BusHandler } from '@sgrud/bus';

const busHandler = new BusHandler();
busHandler.get('io.github.sgrud').subscribe(console.log);
// { handle: 'io.github.sgrud.example', value: 'published' }
```

See

BusHandler

Type parameters

Name	Description
T	Bus type.

Defined in packages/bus/src/handler/handler.ts:61

---

bus.BusValue.handle

handle

• `Readonly` **handle**: '${string}.${string}.${string}'

Emitting BusHandle.

Defined in packages/bus/src/handler/handler.ts:68

---

bus.BusValue.value

value

• `Readonly` **value**: `T`

Emitted `value`.

Defined in packages/bus/src/handler/handler.ts:73

---

bus.BusWorker

BusWorker

• **BusWorker**: `Object`

The **BusWorker** is a Worker process, Spawned by the BusHandler to handle all published and subscribed to busses and the aggregation of their values depending on their hierarchy.

`Decorator`

Thread

`Decorator`

Singleton

`See`

BusHandler

Defined in packages/bus/src/worker/index.ts:23

---

bus.BusWorker.constructor

constructor

• **new BusWorker**()

Public **constructor**. This **constructor** is called once when the BusHandler Spawns the Worker running this class.

Defined in packages/bus/src/worker/index.ts:53

---

bus.BusWorker.get

get

▸ **get**(handle): `Observable<BusValue<any>>`

Invoking this method **get**s the Observable bus represented by the supplied `handle`. This method is called by the BusHandler and is only then proxied to the Worker running this class.

Parameters

Name	Type	Description
`handle`	`${string}.${string}.${string}`	BusHandle to **get**.

Returns `Observable<BusValue<any>>`

Observable bus for `handle`.

Defined in packages/bus/src/worker/index.ts:72

---

bus.BusWorker.set

set

▸ **set**(handle, bus): `void`

Invoking this method **set**s the supplied Observable `bus` for the supplied `handle`. This method is called by the BusHandler and is only then proxied to the Worker running this class.

Parameters

Name	Type	Description
`handle`	`${string}.${string}.${string}`	BusHandle to **set**.
`bus`	`Observable<any>`	Observable bus for `handle`.

Returns `void`

Defined in packages/bus/src/worker/index.ts:99

---

bus.BusWorker.busses

busses

• `Private` `Readonly` **busses**: `Map<${string}.${string}.${string}, Observable<BusValue<any>>>`

Internal mapping containing all established **busses**. Updating this mapping should always be accompanied by an emittance of *changes*.

Defined in packages/bus/src/worker/index.ts:29

---

bus.BusWorker.changes

changes

• `Private` `Readonly` **changes**: `BehaviorSubject<BusWorker>`

BehaviorSubject emitting every time a bus is added or deleted from the internal *busses* mapping, i.e., when **changes** occur on the *busses* mapping. This emittance is used to recompile the open Subscriptions previously obtained to through use of the *get* method.

Defined in

packages/bus/src/worker/index.ts:40

---

bus.Publish

Publish

▸ **Publish**(`handle`, `source?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. This decorator **publish**es the decorated property value under the supplied `handle`. If the supplied `source` isn't an Observable it is assumed to reference a property key of the prototype containing the decorated property. The first instance value assigned to this `source` property is assigned as readonly on the instance and appended to the supplied `handle`, thus creating an *instance-scoped handle*. This *scoped handle* is then used to **publish** the first instance value assigned to the decorated property. This implies that the publication to the underlying bus will wait until both the decorated property and the referenced `source` property are assigned values. If the supplied `source` is of an Observable type, this Observable is **publish**ed under the supplied `handle` and assigned as readonly to the decorated prototype property. If no `source` is supplied, a new Subject will be created and implicitly supplied as `source`. This decorator is more or less the opposite of the Subscribe decorator, while both rely on the BusHandler to fulfill contracts.

Precautions should be taken to ensure completion of the supplied Observable `source` as otherwise memory leaks may occur due to dangling subscriptions.

Example

Publish the `'io.github.sgrud.example'` bus:

```
import type { Subject } from 'rxjs';
import { Publish } from '@sgrud/bus';

export class Publisher {

  @Publish('io.github.sgrud.example')
  public readonly bus!: Subject<any>;

}

Publisher.prototype.bus.complete();
```

Example

Publish the `'io.github.sgrud.example'` bus:

```
import { Publish } from '@sgrud/bus';
import { Subject } from 'rxjs';

export class Publisher {

  @Publish('io.github.sgrud', 'scope')
  public readonly bus: Subject<any> = new Subject<any>();

  public constructor(
    private readonly scope: string
  ) { }

}

const publisher = new Publisher('example');
publisher.bus.complete();
```

See

- BusHandler
- Subscribe

Parameters

Name	Type	Description
`handle`	`'${string}.${string}.${string}'`	BusHandle to **publish**.
`source`	`PropertyKey | Observable<any>`	Property key or Observable.

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/bus/src/handler/publish.ts:76

---

bus.Subscribe

Subscribe

▸ **Subscribe**(`handle`, `source?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. This decorator **subscribe**s to an Observable emitting BusValues originating from the supplied `handle` by assigning it to the decorated property. If no `source` is supplied, this Observable is assigned as readonly to the decorated prototype property. Else the supplied `source` is assumed to be referencing a property key of the prototype containing the decorated property. The first instance value assigned to this `source` property is assigned as readonly on the instance and appended to the supplied `handle`, thus creating an *instance-scoped handle*. This *scoped handle* is then used to create an Observable which is assigned as readonly to the decorated property on the instance. This implies that the decorated property will not be assigned an Observable until the referenced `source` property is assigned an instance value. This decorator is more or less the opposite of the Publish decorator, while both rely on the BusHandler to fulfill contracts.

`Example`

Subscribe to the `'io.github.sgrud.example'` bus:

```
import type { BusValue } from '@sgrud/bus';
import type { Observable } from 'rxjs';
import { Subscribe } from '@sgrud/bus';

export class Subscriber {

  @Subscribe('io.github.sgrud.example')
  public readonly bus!: Observable<BusValue<any>>;

}
```

`Example`

Subscribe to the `'io.github.sgrud.example'` bus:

```
import type { BusValue } from '@sgrud/bus';
import type { Observable } from 'rxjs';
import { Subscribe } from '@sgrud/bus';

export class Subscriber {

  @Subscribe('io.github.sgrud', 'scope')
  public readonly bus!: Observable<BusValue<any>>;
```

```
  public constructor(
    public readonly scope: string
  ) { }

}

const subscriber = new Subscriber('example');
```

See

- BusHandler
- Publish

Parameters

Name	Type	Description
`handle`	‘${string}.${string}.${string}‘	BusHandle to **subscribe** to.
`source?`	`PropertyKey`	Property key.

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/bus/src/handler/subscribe.ts:68

Module: core

core

• **core**: `Object`

`@sgrud/core` - The SGRUD Core Module.

The functions and classes found within this module represent the base upon which the SGRUD client libraries are built. Therefore, most of the code provided within this module does not aim at fulfilling one specific high-level need, but is used and intended to be used as low-level building blocks for downstream projects. This practice is employed throughout the SGRUD client libraries, as all modules depend on this core module. By providing the core functionality within this singular module, all downstream SGRUD modules should be considered opt-in functionality which may be used within projects building upon the SGRUD client libraries.

Defined in packages/core/index.ts:19

---

core.Assign

Assign

T **Assign**<`S`, `T`>: { [K in keyof (S & T)]: K extends keyof S ? S[K] : K extends keyof T ? T[K] : never }

Type helper **assign**ing the own property types of all of the enumerable own properties from a source type to a target type.

`Example`

Assign `valueOf()` to `string`:

```ts
import type { Assign } from '@sgrud/core';

const str = 'Hello world' as Assign<{
  valueOf(): 'Hello world';
}, string>;
```

Type parameters

Name	Description
`S`	Source type.
`T`	Target type.

Defined in

packages/core/src/typing/assign.ts:18

---

core.Factor

Factor

▸ **Factor**<`K`>(`targetFactory`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. Replaces the decorated prototype property with a getter, which looks up the linked instance of a target constructor forwarded-referenced by the `targetFactory`.

`Example`

Factor a service:

```ts
import { Factor } from '@sgrud/core';
import { Service } from './service';

export class ServiceHandler {

  @Factor(() => Service)
  private readonly service!: Service;

}
```

`See`

- Linker
- Target

Type parameters

Name	Type	Description
`K`	extends () => `any`	Constructor type.

Parameters

Name	Type	Description
`targetFactory`	() => `K`	Forward reference to the target constructor.

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/core/src/linker/factor.ts:32

---

core.HttpClient

HttpClient

• **HttpClient**: `Object`

The Singleton **HttpClient** is a thin wrapper around the ajax method. The main function of this wrapper is to pipe all requests through a chain of classes extending the abstract HttpProxy class. Thereby interceptors for various requests can be implemented to, e.g., provide API credentials etc.

`Decorator`

Singleton

`See`

HttpProxy

Defined in packages/core/src/http/client.ts:54

---

core.HttpClient.delete

delete

▸ `Static` **delete**<`T`>(`url`): `Observable<AjaxResponse<T>>`

Fires an HTTP **DELETE** request against the supplied `url` upon subscription.

`Example`

Fire an HTTP **DELETE** request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.delete('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:75

---

core.HttpClient.get

get

▸ `Static` **get**<T>(`url`): `Observable<AjaxResponse<T>>`

Fires an HTTP **GET** request against the supplied `url` upon subscription.

`Example`

Fire an HTTP **GET** request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.get('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:97

---

core.HttpClient.head

head

▸ `Static` **head**<T>(`url`): `Observable<AjaxResponse<T>>`

Fires an HTTP **HEAD** request against the supplied `url` upon subscription.

`Example`

Fire an HTTP **HEAD** request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.head('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:120

---

core.HttpClient.patch

patch

▸ `Static` **patch**<T>(`url`, `body`): `Observable<AjaxResponse<T>>`

Fires an HTTP **PATCH** request against the supplied `url` containing the supplied `body` upon subscription.

`Example`

Fire an HTTP **PATCH** request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.patch('https://example.com', {
  bodyContent: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.
`body`	`unknown`	Request body.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:146

---

core.HttpClient.post

post

▸ `Static` **post**<T>(`url`, `body`): `Observable<AjaxResponse<T>>`

Fires an HTTP **POST** request against the supplied `url` containing the supplied `body` upon subscription.

`Example`

Fire an HTTP **POST** request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.post('https://example.com', {
  bodyContent: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.
`body`	`unknown`	Request body.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:172

---

core.HttpClient.put

put

▸ `Static` **put**<T>(`url`, `body`): `Observable<AjaxResponse<T>>`

Fires an HTTP **PUT** request against the supplied `url` containing the supplied `body` upon subscription.

`Example`

Fire an HTTP **PUT** request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.put('https://example.com', {
  bodyContent: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.
`body`	`unknown`	Request body.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:198

---

core.HttpClient.constructor

constructor

• **new HttpClient**()

---

core.HttpClient.handle

handle

▸ **handle**<T>(request): Observable<AjaxResponse<T>>

Generic **handle** method, enforced by the HttpHandler interface. Main method of the this class. Internally pipes the `request` through all linked classes extending HttpProxy.

`Example`

Fire an HTTP custom request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.prototype.handle({
  method: 'GET',
  url: 'https://example.com',
  headers: { 'x-example': 'value' }
}).subscribe(console.log);
```

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
request	AjaxConfig	Requesting AjaxConfig.

Returns Observable<AjaxResponse<T>>

Observable of the requested AjaxResponse.

Implementation of HttpHandler.handle

Defined in packages/core/src/http/client.ts:229

---

core.HttpHandler

HttpHandler

• **HttpHandler**: Object

The **HttpHandler** interface enforces the generic *handle* method with ajax compliant typing on the implementing class or object. This contract is used by the HttpProxy to type the next hops in the HttpClient proxy chain.

`See`

HttpClient

Defined in packages/core/src/http/client.ts:19

---

core.HttpHandler.handle

handle

▸ **handle**<T>(request): Observable<AjaxResponse<T>>

Generic **handle** method enforcing ajax compliant typing. The method signature corresponds to that of the ajax method itself.

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
request	AjaxConfig	Requesting AjaxConfig.

Returns Observable<AjaxResponse<T>>

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/client.ts:34

---

core.HttpProxy

HttpProxy

• Abstract **HttpProxy**: Object

Abstract **HttpProxy** base class to implement proxies, i.e., HTTP request interceptors, on the client side. By extending this abstract base class and providing the extending class to the Linker, e.g., by Targeting it, the respective classes *proxy* method will be called with the request details (which could have been modified by a previous **HttpProxy**) and the next HttpHandler (which could be the next **HttpProxy** or the ajax method), whenever a request is fired through the HttpClient.

Decorator

Provide

Example

Simple **HttpProxy** intercepting `file:` requests:

```
import type { HttpHandler, HttpProxy } from '@sgrud/core';
import type { AjaxConfig, AjaxResponse } from 'rxjs/ajax';
import { Provider, Target } from '@sgrud/core';
import { Observable, of } from 'rxjs';
import { file } from './file';

@Target<typeof FileProxy>()
export class FileProxy
  extends Provider<typeof HttpProxy>('sgrud.core.http.HttpProxy') {

  public override proxy<T>(
    request: AjaxConfig,
    handler: HttpHandler
  ): Observable<AjaxResponse<T>> {
    if (request.url.startsWith('file:')) {
      return of<AjaxResponse<T>>(file);
    }

    return handler.handle<T>(request);
  }
}
```

See

HttpClient

Defined in packages/core/src/http/proxy.ts:53

---

core.HttpProxy.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.core.http.HttpProxy"`

Magic string by which this class is provided.

See

provide

Defined in packages/core/src/http/proxy.ts:62

---

core.HttpProxy.constructor

constructor

• **new HttpProxy**()

---

core.HttpProxy.proxy

proxy

▸ `Abstract` **proxy**<T>(`request`, `handler`): `Observable<AjaxResponse<T>>`

The **proxy** method of linked classes extending the HttpProxy base class is called whenever a request is fired through the HttpClient. The extending class can either pass the `request` to the next `handler`, with or without modifying it, or an interceptor can chose to completely handle a `request` by itself through returning an Observable AjaxResponse.

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	Requesting AjaxConfig.
`handler`	`HttpHandler`	Next HttpHandler.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Defined in packages/core/src/http/proxy.ts:84

---

core.HttpState

HttpState

• **HttpState**: `Object`

The Targeted Singleton **HttpState** is a built-in HttpProxy intercepting all requests fired through the Http-Client. This proxy implements the observable pattern, through which it emits an array of all currently open connections every time a new request is fired or a previously fired request completes.

`Decorator`

Target

`Decorator`

Singleton

`See`

- HttpClient
- HttpProxy

Defined in packages/core/src/http/state.ts:30

---

core.HttpState.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.core.http.HttpProxy"`

Magic string by which this class is provided.

`See`

provide

Inherited from HttpProxy.[provide]

Defined in packages/core/src/http/proxy.ts:62

---

core.HttpState.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<AjaxResponse<any>[]>`

Type declaration ▸ (): `Subscribable<AjaxResponse<any>[]>`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits an array of all active requests whenever this list changes. Using the returned Subscribable, e.g., a load indicator can easily be implemented.

`Example`

Subscribe to the currently active requests:

```
import { HttpState, Linker } from '@sgrud/core';
import { from } from 'rxjs';

const httpState = new Linker<typeof HttpState>().get(HttpState);
from(httpState).subscribe(console.log);
```

Returns `Subscribable<AjaxResponse<any>[]>`

Callback to a Subscribable.

Defined in packages/core/src/http/state.ts:53

---

core.HttpState.constructor

constructor

• **new HttpState**()

Public **constructor**. Called by the Target decorator to link this HttpProxy into the proxy chain.

Overrides HttpProxy.constructor

Defined in packages/core/src/http/state.ts:90

---

core.HttpState.proxy

proxy

▸ **proxy**<T>(`request`, `handler`): `Observable<AjaxResponse<T>>`

Overridden **proxy** method of the HttpProxy base class. Mutates the `request` to also emit progress events while the it is running. These progress events will be consumed by the HttpState interceptor and re-supplied via the Subscribable returned by the interop getter.

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	Requesting AjaxConfig.
`handler`	`HttpHandler`	Next HttpHandler.

Returns `Observable<AjaxResponse<T>>`

Observable of the requested AjaxResponse.

Overrides HttpProxy.proxy

Defined in packages/core/src/http/state.ts:116

---

core.HttpState.changes

changes

• `Private` `Readonly` **changes**: `BehaviorSubject<HttpState>`

BehaviorSubject emitting every time a request is added to or deleted from the internal *running* mapping.

Defined in packages/core/src/http/state.ts:61

---

core.HttpState.running

running

• `Private` `Readonly` **running**: `Map<AjaxConfig, AjaxResponse<any>>`

Internal mapping containing all running requests. Updating this map should always be accompanied by an emittance of the *changes* BehaviorSubject.

Defined in packages/core/src/http/state.ts:69

---

core.Kernel

Kernel

• **Kernel**: `Object`

Singleton **Kernel** class. The **Kernel** is essentially a dependency loader for ESM bundles (and their respective `importmaps`) or, depending on the runtime context and capabilities, UMD bundles and their transitive dependencies. By making use of the **Kernel**, applications based on the SGRUD client libraries may be comprised of multiple, optionally loaded Modules, which, depending on the application structure and configuration, can be loaded initially, by supplying them as dependencies through the corresponding API endpoint (which can be customized through the second parameter to the *constructor*), or later on, manually.

`Decorator`

Singleton

`See`

Module

Defined in packages/core/src/kernel/kernel.ts:17

packages/core/src/kernel/kernel.ts:183

---

core.Kernel.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<Module>`

Type declaration ▸ (): `Subscribable<Module>`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits every Module that is successfully loaded.

`Example`

Subscribe to the stream of loaded Modules:

```
import { Kernel } from '@sgrud/core';
import { from } from 'rxjs';

from(new Kernel()).subscribe(console.log);
```

Returns `Subscribable<Module>`

Callback to a Subscribable.

Defined in packages/core/src/kernel/kernel.ts:203

---

core.Kernel.baseHref

baseHref

• `Readonly` **baseHref**: `string` = `location.origin`

Base href for building, e.g., the *endpoint* and *nodeModule* URLs.

`Default Value`

`location.origin`

Defined in packages/core/src/kernel/kernel.ts:297

---

core.Kernel.constructor

constructor

• **new Kernel**(`baseHref?`, `endpoint?`, `nodeModules?`)

Singleton **constructor**. The first time, this **constructor** is called, it will retrieve the list of modules which should be loaded and then call *insmod* on all those modules and their transitive dependencies. Every subsequent **constructor** call will ignore all arguments and return the Singleton instance. Through subscribing to the Subscribable returned by the observable interop getter, the initial Module loading progress can be tracked.

`Example`

Instantiate the **Kernel**:

```
import { Kernel } from '@sgrud/core';

const kernel = new Kernel(
  'https://example.com',
  '/context/api/sgrud',
  'https://unpkg.com'
);
```

Parameters

Name	Type	Default value	Description
`baseHref`	`string`	`location.origin`	Base href for building URLs.
`endpoint`	`string`	`undefined`	Href of the SGRUD API endpoint.
`nodeModules`	`string`	`undefined`	Href to load node modules from.

Defined in packages/core/src/kernel/kernel.ts:290

---

core.Kernel.endpoint

endpoint

• `Readonly` **endpoint**: `string`

Href of the SGRUD API **endpoint**. Modules to be initially loaded (by their names) are requested from the URL `${endpoint}/insmod` when this class is constructed for the first time.

`Default Value`

```
baseHref + '/api/sgrud/v1'
```

Defined in packages/core/src/kernel/kernel.ts:309

---

core.Kernel.insmod

insmod

▸ **insmod**(`module`, `source?`, `entryModule?`): `Observable<Module>`

Insert modules. Calling this method while supplying a valid `module` definition will chain the `module` dependencies and the `module` itself into an Observable, which is then returned. When multiple `modules` are inserted, their dependencies are deduplicated by internally tracking all `modules` and their transitive dependencies as separate *loaders*. Depending on the browser context, either the UMD or ESM bundles (and their respective `importmaps`) are loaded via calling the *script* method. When inserting Modules which

contain transitive *sgrudDependencies*, their compatibility is checked. Should a dependency version mismatch, the returned Observable will throw.

Throws

Observable of RangeError or ReferenceError.

Example

Insert a module by definition:

```
import { Kernel } from '@sgrud/core';
import packageJson from 'module/package.json';

new Kernel().insmod(packageJson).subscribe(console.log);
```

Parameters

Name	Type	Default value	Description
`module`	`Module`	`undefined`	Module definition.
`source`	`string`	`undefined`	Optional Module source.
`entryModule`	`boolean`	`false`	Wether to run the Module.

Returns `Observable<Module>`

Observable of the Module loading.

Defined in packages/core/src/kernel/kernel.ts:372

---

core.Kernel.nodeModules

nodeModules

• `Readonly` **nodeModules**: `string`

Href to load node modules from. All JavaScript assets belonging to packages installed via NPM should be located here.

Default Value

```
baseHref + '/node_modules'
```

Defined in packages/core/src/kernel/kernel.ts:317

---

core.Kernel.resolve

resolve

▸ **resolve**(`name`, `source?`): `Observable<Module>`

Resolves a Module definition by its `name`. The Module `name` is appended to the *nodeModules* path and the package.json file therein retrieved via HTTP GET. The parsed package.json is then emitted by the returned Observable.

Example

Resolve a Module definition:

```
import { Kernel } from '@sgrud/core';

new Kernel().resolve('module').subscribe(console.log);
```

Parameters

Name	Type	Description
`name`	`string`	Module name.
`source`	`string`	Optional Module source.

Returns `Observable<Module>`

Observable of the Module definition.

Defined in packages/core/src/kernel/kernel.ts:501

---

core.Kernel.script

script

▸ **script**(`props`): `Observable<void>`

Inserts an HTMLScriptElement and applies the supplied `props` to it. The returned Observable emits and completes when the *onload* handler is called on the HTMLScriptElement. If no external *src* is supplied through the `props`, the *onload* handler is called asynchronously. When the returned Observable completes, the inserted HTMLScriptElement is removed.

`Example`

Insert an HTMLScriptElement:

```
import { Kernel } from '@sgrud/core';

new Kernel().script({
  src: '/node_modules/module/bundle.js',
  type: 'text/javascript'
}).subscribe();
```

Parameters

Name	Type	Description
`props`	`Partial<HTMLScriptElement>`	HTMLScriptElement properties.

Returns `Observable<void>`

Observable of the HTMLScriptElements load and removal.

Defined in packages/core/src/kernel/kernel.ts:539

---

core.Kernel.verify

verify

▸ **verify**(`props`): `Observable<void>`

Inserts an HTML link element and applies the supplied `props` to it. This method should be used to **verify** a Module bundle before importing and evaluating it, by providing its Subresource Integrity.

`Example`

Verify the Subresource Integrity:

```
import { Kernel } from '@sgrud/core';

new Kernel().verify({
  href: '/node_modules/module/index.js',
  integrity: 'sha256-[...]',
  rel: 'modulepreload'
}).subscribe();
```

Parameters

Name	Type	Description
`props`	`Partial<HTMLLinkElement>`	Link element properties.

Returns `Observable<void>`

Observable of link appendage and removal.

Defined in packages/core/src/kernel/kernel.ts:584

---

core.Kernel.imports

imports

• `Private` `Readonly` **imports**: `Map<string, string>`

Internal mapping of all via `importmaps` defined Module identifiers to their corresponding paths. This mapping is used for housekeeping, e.g., to prevent the same Module identifier to be defined multiple times.

Defined in packages/core/src/kernel/kernel.ts:212

---

core.Kernel.loaders

loaders

• `Private` `Readonly` **loaders**: `Map<string, ReplaySubject<Module>>`

Internal mapping of all Modules **loaders** to a ReplaySubject. This ReplaySubject tracks the Module loading process as such, that it emits the Module definition once the respective Module is fully loaded (including dependencies etc.) and then completes.

Defined in packages/core/src/kernel/kernel.ts:223

---

core.Kernel.loading

loading

• `Private` `Readonly` **loading**: `ReplaySubject<Module>`

Internal ReplaySubject tracking the **loading** state of Modules. An Observable form of this ReplaySubject may be retrieved by subscribing to the Subscribable returned by the interop getter. The internal ReplaySubject (and the retrievable Observable) emits all Module definitions loaded throughout the lifespan of this class.

Defined in packages/core/src/kernel/kernel.ts:237

---

core.Kernel.shimmed

shimmed

• `Private` `Readonly` **shimmed**: `string`

Internally used string to suffix the `importmap` and `module` types of HTMLScriptElements with, if applicable. This string is set to whatever trails the type of HTMLScriptElements encountered upon initialization, iff their type starts with `importmap`.

Defined in packages/core/src/kernel/kernel.ts:247

---

core.Kernel

Kernel

• **Kernel**: `Object`

Kernel namespace containing types and interfaces used and intended to be used in conjunction with the Singleton Kernel class.

See

Kernel

Defined in packages/core/src/kernel/kernel.ts:17

packages/core/src/kernel/kernel.ts:183

---

core.Kernel.Digest

Digest

T **Digest**: ‘sha${256 | 384 | 512}-${string}‘

String literal helper type. Enforces any assigned string to represent a browser-parsable **Digest** hash.

Example

A valid **Digest**:

```
import type { Digest } from '@sgrud/core';

const digest: Digest = 'sha256-[...]';
```

Defined in packages/core/src/kernel/kernel.ts:31

---

core.Kernel.Module

Module

• **Module**: `Object`

Interface describing the shape of a **Module** while being aligned with well-known package.json fields. This interface additionally specifies optional `sgrudDependencies` and `webDependencies` mappings, which both are used by the Kernel to determine SGRUD module dependencies and runtime (web) dependencies.

Example

An exemplary **Module** definition:

```
import type { Kernel } from '@sgrud/core';

const module: Kernel.Module = {
  name: 'module',
  version: '0.0.0',
  exports: './module.exports.js',
  unpkg: './module.unpkg.js',
  sgrudDependencies: {
    sgrudDependency: '^0.0.1'
  },
  webDependencies: {
    webDependency: {
      exports: {
        webDependency: './webDependency.exports.js'
      },
      unpkg: [
        './webDependency.unpkg.js'
      ]
```

```
    }
  }
};
```

Defined in packages/core/src/kernel/kernel.ts:70

Kernel.Module.digest

digest

• `Optional` `Readonly` **digest**: `Record<string, 'sha256-${string}' | 'sha384-${string}' | 'sha512-${string}'>`

Optional bundle Digests. If hashes are supplied, they will be used to verify the Subresource Integrity of the respective bundles.

Defined in packages/core/src/kernel/kernel.ts:101

Kernel.Module.exports

exports

• `Optional` `Readonly` **exports**: `string`

ESM entry point.

Defined in packages/core/src/kernel/kernel.ts:87

Kernel.Module.name

name

• `Readonly` **name**: `string`

Name of the **Module**.

Defined in packages/core/src/kernel/kernel.ts:75

Kernel.Module.sgrudDependencies

sgrudDependencies

• `Optional` `Readonly` **sgrudDependencies**: `Record<string, string>`

Optional SGRUD dependencies.

Defined in packages/core/src/kernel/kernel.ts:108

Kernel.Module.unpkg

unpkg

• `Optional` `Readonly` **unpkg**: `string`

UMD entry point.

Defined in packages/core/src/kernel/kernel.ts:92

Kernel.Module.version

version

• `Readonly` **version**: `string`

Module version, formatted as semver.

Defined in

packages/core/src/kernel/kernel.ts:82

Kernel.Module.webDependencies

webDependencies

• `Optional` `Readonly` **webDependencies**: `Record<string, WebDependency>`

Optional WebDependency mapping.

Defined in

packages/core/src/kernel/kernel.ts:115

---

core.Kernel.WebDependency

WebDependency

• **WebDependency**: `Object`

Interface describing runtime dependencies of a Module. A Module may specify an array of UMD bundles to be loaded by the Kernel through the `unpkg` property. A Module may also specify a mapping of import specifiers to Module-relative paths through the `exports` property. Every specified **WebDependency** is loaded before respective bundles of the Module, which depends on the specified **WebDependency**, will be loaded themselves.

`Example`

An exemplary **webDependency** definition:

```
import type { Kernel } from '@sgrud/core';

const webDependency: Kernel.WebDependency = {
  exports: {
    webDependency: './webDependency.exports.js'
  },
  unpkg: [
    './webDependency.unpkg.js'
  ]
};
```

Defined in

packages/core/src/kernel/kernel.ts:147

Kernel.WebDependency.exports

exports

• `Optional` `Readonly` **exports**: `Record<string, string>`

Optional ESM runtime dependencies.

Defined in

packages/core/src/kernel/kernel.ts:152

Kernel.WebDependency.unpkg

unpkg

• `Optional` `Readonly` **unpkg**: `string[]`

Optional UMD runtime dependencies.

Defined in packages/core/src/kernel/kernel.ts:157

---

core.Linker

Linker

• **Linker**<K, V>: `Object`

The Singleton **Linker** class provides the means to lookup instances of Targeted constructors. The **Linker** is used throughout the SGRUD client libraries, e.g., by the Factor decorator, to provide and retrieve different centrally provisioned class instances. To programmatically insert some links, the inherited *constructor* or *set* methods can be used. The former will insert all entries into this Singleton link mapping, internally calling the latter for each.

`Decorator`

Singleton

`Example`

Preemptively link an instance:

```
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>([
  [Service, new Service('linked')]
]);
```

Type parameters

Name	Type	Description
K	extends () => V	Constructor type.
V	InstanceType<K>	Instance type.

Defined in packages/core/src/linker/linker.ts:41

---

core.Linker.constructor

constructor

• **new Linker**<K, V>(`entries?`)

Type parameters

Name	Type
K	extends () => V
V	InstanceType<K>

Parameters

Name	Type
entries?	null | readonly readonly [K, V][]

Inherited from Map<K, V>.constructor

Defined in node_modules/typescript/lib/lib.es2015.collection.d.ts:53

• **new Linker**<K, V>(`iterable?`)

Type parameters

Name	Type
K	extends () => V
V	InstanceType<K>

Parameters

Name	Type
iterable?	null | Iterable<readonly [K, V]>

Inherited from Map<K, V>.constructor

Defined in node_modules/typescript/lib/lib.es2015.iterable.d.ts:161

---

core.Linker.get

get

▸ **get**(target): V

Overridden **get** method. Calling this method looks up the linked instance based on the supplied target constructor. If no linked instance is found, one is created by calling the new operator on the target constructor. Therefor the target constructors must not require parameters.

Example

Retrieve a linked instance:

```
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>().get(Service);
```

Parameters

Name	Type	Description
target	K	Target constructor.

Returns V

Linked instance.

Overrides Map.get

Defined in packages/core/src/linker/linker.ts:64

---

core.Linker.getAll

getAll

▸ **getAll**(target): V[]

Returns all linked instances, which satisfy instanceof target. Use this method when multiple linked target constructors extend the same base class and are to be retrieved.

Example

Retrieve all linked instances:

```ts
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>().getAll(Service);
```

Parameters

Name	Type	Description
`target`	`K`	Target constructor.

Returns `V[]`

All linked instances.

Defined in packages/core/src/linker/linker.ts:92

---

core.Merge

Merge

T **Merge**<`T`>: `T` extends `T` ? (_: `T`) => `T` : `never` extends (_: infer I) => `T` ? `I` : `never`

Type helper to convert union types (`A | B`) to intersection types (`A & B`).

`Remarks`

https://github.com/microsoft/TypeScript/issues/29594

Type parameters

Name	Description
`T`	Union type.

Defined in packages/core/src/typing/merge.ts:8

---

core.Mutable

Mutable

T **Mutable**<`T`>: { -readonly [K in keyof T]: T[K] }

Type helper marking the supplied type as **Mutable** (opposed to *readonly*).

`Remarks`

https://github.com/Microsoft/TypeScript/issues/24509

Type parameters

Name	Type	Description
`T`	extends `object`	Readonly type.

Defined in packages/core/src/typing/mutable.ts:8

---

core.Provide

Provide

`T` **Provide**`<K, V>: (...args: any[]) => InstanceType<V> & { [provide]: K extends Registration ? K : Registration }`

Type helper enforcing the provide symbol property containing a magic string (typed as Registration) on base constructors decorated with the corresponding Provide decorator. The **Provide** type helper is also used by the Provider decorator.

See

Provide

Type parameters

Name	Type	Description
`K`	extends `Registration`	Magic string type.
`V`	extends `(...args: any[]) => InstanceType<V>`	Constructor type.

Defined in packages/core/src/super/provide.ts:78

packages/core/src/super/provide.ts:26

---

core.Provide

Provide

▸ **Provide**`<V, K>(): (constructor: V) => void`

Class decorator factory. **Provide**s the decorated class to extending classes. Applying the **Provide** decorator enforces the Provide type which entails the declaration of a static provide property typed as Registration. The magic string assigned to this static property is used by the Provider factory function to lookup base classes within the Registry mapping.

Example

Provide a base class:

```
import { Provide, provide } from '@sgrud/core';

@Provide<typeof Base>()
export abstract class Base {

  public static readonly [provide]:
  'sgrud.example.Base' = 'sgrud.example.Base' as const;

}
```

See

- Provider
- Registry

Type parameters

Name	Type	Description
`V`	extends `Provide<K, V>`	Constructor type.
`K`	extends ‘${string}.${string}.${string}‘ = `V[typeof provide]`	Magic string type.

Returns `fn`

Class decorator.

▸ `(constructor): void`

Parameters

Name	Type
`constructor`	`V`

Returns `void`

Defined in packages/core/src/super/provide.ts:78

---

core.Provider

Provider

▸ **Provider**<`V`, `K`>(`provider`): `V`

Provider of base classes. Extending this mixin-style function while supplying the `typeof` a Provided constructor enforces type safety and hinting on the supplied magic string and the resulting class which `extends` this **Provider** mixin. The main purpose of this pattern is bridging module gaps by de-coupling bundle files while maintaining a well-defined prototype chain. This still requires the base class to be defined (and Provided) before extension but allows intellisense'd OOP patterns across multiple modules while maintaining runtime language specifications.

`Example`

Extend a provided class:

```
import type { Base } from 'example-module';
import { Provider } from '@sgrud/core';

export class Class
  extends Provider<typeof Base>('org.example.Base') {

  public constructor(...args: any[]) {
    super(...args);
  }

}
```

`See`

- Provide
- Registry

Type parameters

Name	Type	Description
`V`	extends `Provide<K, V>`	Constructor type.
`K`	extends '${string}.${string}.${string}' = `V[typeof provide]`	Magic string type.

Parameters

Name	Type	Description
`provider`	`K`	Magic string.

Returns `V`

Providing constructor.

Defined in packages/core/src/super/provider.ts:69

---

core.Provider

Provider

• **Provider**<V>: `Object`

Type helper to allow referencing Provided constructors as `new`-able targets. Used and intended to be used in conjunction with the Provider decorator.

See

Provider

Type parameters

Name	Description
V	Instance type.

Defined in packages/core/src/super/provider.ts:69

packages/core/src/super/provider.ts:16

---

core.Provider.[provide]

[provide]

• `Readonly` **[provide]**: '${string}.${string}.${string}'

Enforced provider contract.

Defined in packages/core/src/super/provider.ts:21

---

core.Provider.constructor

constructor

• **constructor**: `Object`

---

core.Provider.constructor

constructor

• **new Provider**(...`args`)

Enforced constructor contract.

Parameters

Name	Type	Description
...args	any[]	Class constructor rest parameter.

Defined in packages/core/src/super/provider.ts:28

---

core.Registration

Registration

T **Registration**: ‘${string}.${string}.${string}‘

String literal helper type. Enforces any assigned string to contain at least three dots. **Registration**s are used by the Registry to alias classes extending the base Provider as magic strings and should represent sane module paths in dot-notation.

`Example`

Library-wide **Registration** pattern:

```ts
import type { Registration } from '@sgrud/core';

const registration: Registration = 'sgrud.module.path.ClassName';
```

`See`

Registry

Defined in

packages/core/src/super/registry.ts:22

---

core.Registry

Registry

• **Registry**<K, V>: `Object`

The Singleton **Registry** is a mapping used by the Provider to lookup Provided constructors by Registrations upon class extension. Magic strings should represent sane module paths in dot-notation. To programmatically provide constructors by Registrations to extending classes, the inherited *constructor* or *set* methods are available. The former will insert all entries into this Singleton **Registry** map, internally calling the latter for each. Whenever a currently not registered constructor is requested, an intermediary class is created, *cached* internally and returned. When the actual constructor is registered later, the previously created intermediary class is removed from the internal caching and further steps are taken to guarantee the transparent addressing of the actual constructor through the dropped intermediary class.

`Decorator`

Singleton

`See`

- Provide
- Provider

Type parameters

Name	Type	Description
K	extends Registration	Magic string type.
V	extends (...args: any[]) => InstanceType<V>	Constructor type.

Defined in

packages/core/src/super/registry.ts:59

---

core.Registry.constructor

constructor

• **new Registry**<K, V>(tuples?)

Public **constructor**. The constructor of this class accepts the same parameters as its overridden `super` **constructor** and acts the same. I.e., through instantiating this Singleton class and passing a list of tuples of Registrations and their corresponding constructors, these tuples will be stored.

`Example`

Preemptively provide a constructor by magic string:

```ts
import type { Registration } from '@sgrud/core';
import { Registry } from '@sgrud/core';
import { Service } from './service';

new Registry<Registration, typeof Service>([
  ['sgrud.example.Service', Service]
]);
```

Type parameters

Name	Type
K	extends `${string}.${string}.${string}`
V	extends (...args: any[]) => InstanceType<V>

Parameters

Name	Type	Description
tuples?	[K, V][]	List of constructors to provide.

Overrides Map<K, V>.constructor

Defined in packages/core/src/super/registry.ts:109

---

core.Registry.get

get

▸ **get**(registration): V

Overridden **get** method. Looks up the Provided constructor by magic string. If no provided constructor is found, an intermediary class is created, *cached* internally and returned. While this intermediary class and the functionality supporting it takes care of inheritance, i.e., allows to forward-reference base classes to be extended, it cannot substitute for the actual extended constructor. Therefore, static extension of forward-referenced classes may be used, but as long as the actual extended constructor is not registered (and therefore the intermediary class is still acting as inheritance cache), the extending class cannot be instantiated, called etc. Doing so will result in a ReferenceError being thrown.

Throws

ReferenceError.

Example

Retrieve a provided constructor by magic string:

```ts
import type { Registration } from '@sgrud/core';
import type { Service } from 'example-module';
import { Registry } from '@sgrud/core';

new Registry<Registration, typeof Service>().get('org.example.Service');
```

Parameters

Name	Type	Description
registration	K	Magic string.

Returns V

Provided constructor.

Overrides Map.get

Defined in packages/core/src/super/registry.ts:151

---

core.Registry.set

set

▸ **set**(`registration`, `constructor`): `Registry<K, V>`

Overridden **set** method. Whenever a constructor is provided by magic string through calling this method, a check is run, wether this constructor was previously requested and therefore was *cached* as intermediary class. If so, the intermediary class is removed from this internal mapping and further steps are taken to guarantee the transparent addressing of the newly provided constructor through the previously *cached* and now dropped intermediary class.

`Example`

Preemptively provide a constructor by magic string:

```
import type { Registration } from '@sgrud/core';
import { Registry } from '@sgrud/core';
import { Service } from './service';

new Registry<Registration, typeof Service>().set(
  'org.example.Service',
  Service
);
```

Parameters

Name	Type	Description
`registration`	`K`	Magic string.
`constructor`	`V`	Providing constructor.

Returns `Registry<K, V>`

This instance.

Overrides Map.set

Defined in packages/core/src/super/registry.ts:207

---

core.Registry.cached

cached

• `Private` `Readonly` **cached**: `Map<K, V>`

Internally used mapping of all **cached**, i.e., forward-referenced, constructors. Whenever a constructor, which is not currently registered, is requested as a Provider, an intermediary class is created and stored within this map until the actual constructor is registered. As soon as this happens, the intermediary class is removed from this map and further steps are taken to guarantee the transparent addressing of the actual constructor through the dropped intermediary class.

Defined in packages/core/src/super/registry.ts:75

---

core.Registry.caches

caches

• `Private` `Readonly` **caches**: `WeakSet<V>`

Internally used (weak) set containing all intermediary classes created upon requesting a currently not registered constructor as provider. This (weak) set is used internally to check, if a intermediary class has already been replaced by the actual constructor.

Defined in packages/core/src/super/registry.ts:83

---

core.Singleton

Singleton

▸ **Singleton**<T>(apply?): (constructor: T) => T

Class decorator factory. Enforces a transparent **Singleton** pattern on the decorated class. When calling the new operator on a decorated class for the first time, an instance of the decorated class is created using the supplied arguments, if any. This instance will remain the **Singleton** instance of the decorated class indefinitely. When calling the new operator on a decorated class already instantiated, the **Singleton** pattern is enforced and the previously constructed instance is returned. Instead, if provided, the apply callback is fired with the **Singleton** instance and the new invocation parameters.

Example

Singleton class:

```
import { Singleton } from '@sgrud/core';

@Singleton<typeof Service>()
export class Service { }

new Service() === new Service(); // true
```

Type parameters

Name	Type	Description
T	extends (...args: any[]) => any	Constructor type.

Parameters

Name	Type	Description
apply?	(self: InstanceType<T>, args: ConstructorParameters<T>) => InstanceType<T>	Construct function.

Returns fn

Class decorator.

▸ (constructor): T

Parameters

Name	Type
constructor	T

Returns T

Defined in packages/core/src/utility/singleton.ts:27

---

core.Spawn

Spawn

▸ **Spawn**(`worker`, `source?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

This prototype property decorator factory **Spawn**s a Worker, wraps it with Comlink and assigns it to the decorated prototype property.

`Example`

Spawn a Worker:

```
import { Spawn, Thread } from '@sgrud/core';
import { ExampleWorker } from 'example-worker';

export class ExampleWorkerHandler {

  @Spawn('example-worker')
  private static readonly worker: Thread<ExampleWorker>;

}
```

`See`

Thread

Parameters

Name	Type	Description
`worker`	`string` | `Endpoint` | `NodeEndpoint`	Worker constructor to **Spawn**.
`source?`	`string`	Optional Module source.

Returns `fn`

Class property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/core/src/thread/spawn.ts:39

---

core.Target

Target

▸ **Target**<`K`>(`factoryArgs?`, `target?`): (`constructor`: `K`) => `void`

Class decorator factory. Links the **Target**ed constructor to its corresponding instance by applying the supplied `factoryArgs`. Employ this helper to link **Target**ed constructors with required arguments. Supplying a `target` constructor overrides its linked instance, if any, with the constructed instance.

`Example`

Target a service:

```
import { Target } from '@sgrud/core';

@Target<typeof Service>(['default'])
export class Service {

  public constructor(
```

```
    public readonly param: string
  ) { }

}
```

Example

Factor a **Target**ed service:

```
import type { Target } from '@sgrud/core';
import { Factor } from '@sgrud/core';
import { Service } from './service';

export class ServiceHandler {

  @Factor<Target<Service>>(() => Service)
  public readonly service!: Service;

}
```

See

- Factor
- Linker

Type parameters

Name	Type	Description
K	extends (...args: any[]) => any	Constructor type.

Parameters

Name	Type	Description
factoryArgs?	ConstructorParameters<K>	**Target** constructor arguments.
target?	K	**Target** constructor override.

Returns `fn`

Class decorator.

▸ (constructor): void

Parameters

Name	Type
constructor	K

Returns `void`

Defined in packages/core/src/linker/target.ts:71

---

core.Target

Target

• **Target**<V>: Object

Type helper to allow Factoring **Target**ed constructors with required arguments. Used and to be used in conjunction with the Target decorator.

Type parameters

Name	Description
V	Instance type.

Defined in packages/core/src/linker/target.ts:71

packages/core/src/linker/target.ts:12

---

core.Target.constructor

constructor

• **constructor**: Object

---

core.Target.constructor

constructor

• **new Target**(...args)

Enforced constructor contract.

Parameters

Name	Type	Description
...args	any[]	Constructor arguments.

Defined in packages/core/src/linker/target.ts:19

---

core.Thread

Thread

T **Thread**<T>: Promise<Remote<T>>

Type alias describing an exposed class in a remote context. Created by wrapping a Comlink *Remote* in a *Promise*. Used and intended to be used in conjunction with the Thread decorator.

See

Thread

Type parameters

Name	Description
T	Instance type.

Defined in packages/core/src/thread/thread.ts:41

packages/core/src/thread/thread.ts:18

---

core.Thread

Thread

▸ **Thread**(): (`constructor`: () => `any`) => `void`

Class decorator factory. Exposes an instance of the decorated class as Worker **Thread** via Comlink.

Example

ExampleWorker **Thread**:

```ts
import { Thread } from '@sgrud/core';

@Thread()
export class ExampleWorker { }
```

See

Spawn

Returns `fn`

Class decorator.

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	() => `any`

Returns `void`

Defined in packages/core/src/thread/thread.ts:41

---

core.TypeOf

TypeOf

• `Abstract` **TypeOf**: `Object`

Strict type-assertion and runtime type-checking utility. When type-checking variables in the global scope, e.g., `window` or `process`, make use of the `globalThis` object.

Example

Type-check global context:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.process(globalThis.process); // true if running in node context
TypeOf.window(globalThis.window);   // true if running in browser context
```

Defined in packages/core/src/utility/type-of.ts:15

---

core.TypeOf.array

array

▸ `Static` **array**(`value`): value is any[]

Type-check `value` for `Array<any>`.

Example

Type-check `null` for `Array<any>`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.array(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is any[]

Whether `value` is of type `Array<any>`.

Defined in packages/core/src/utility/type-of.ts:31

---

core.TypeOf.boolean

boolean

▸ `Static` **boolean**(`value`): value is boolean

Type-check `value` for `boolean`.

`Example`

Type-check `null` for `boolean`:

```
import { TypeOf } from '@sgrud/core';

TypeOf.boolean(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is boolean

Whether `value` is of type `boolean`.

Defined in packages/core/src/utility/type-of.ts:49

---

core.TypeOf.date

date

▸ `Static` **date**(`value`): value is Date

Type-check `value` for `Date`.

`Example`

Type-check `null` for `Date`:

```
import { TypeOf } from '@sgrud/core';

TypeOf.date(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Date

Whether `value` is of type `Date`.

Defined in packages/core/src/utility/type-of.ts:67

---

core.TypeOf.function

function

▸ `Static` **function**(`value`): value is Function

Type-check `value` for `Function`.

`Example`

Type-check `null` for `Function`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.function(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Function

Whether `value` is of type `Function`.

Defined in packages/core/src/utility/type-of.ts:85

---

core.TypeOf.global

global

▸ `Static` **global**(`value`): value is typeof globalThis

Type-check `value` for `global`.

`Example`

Type-check `null` for `typeof globalThis`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.global(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is typeof globalThis

Whether `value` is of type `typeof globalThis`.

Defined in packages/core/src/utility/type-of.ts:103

---

core.TypeOf.null

null

▸ `Static` **null**(`value`): value is null

Type-check `value` for `null`.

`Example`

Type-check `null` for `null`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.null(null); // true
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is null

Whether `value` is of type `null`.

Defined in packages/core/src/utility/type-of.ts:121

---

core.TypeOf.number

number

▸ `Static` **number**(`value`): value is number

Type-check `value` for `number`.

`Example`

Type-check `null` for `number`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.number(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is number

Whether `value` is of type `number`.

Defined in packages/core/src/utility/type-of.ts:139

---

core.TypeOf.object

object

▸ `Static` **object**(`value`): value is object

Type-check `value` for `object`.

`Example`

Type-check `null` for `object`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.object(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is object

Whether `value` is of type `object`.

Defined in packages/core/src/utility/type-of.ts:157

---

core.TypeOf.process

process

▸ `Static` **process**(`value`): value is Process

Type-check `value` for `NodeJS.Process`.

`Example`

Type-check `null` for `NodeJS.Process`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.process(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Process

Whether `value` is of type `NodeJS.Process`.

Defined in packages/core/src/utility/type-of.ts:175

---

core.TypeOf.promise

promise

▸ `Static` **promise**(`value`): value is Promise<any>

Type-check `value` for `Promise<any>`.

`Example`

Type-check `null` for `Promise<any>`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.promise(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Promise<any>

Whether `value` is of type `Promise<any>`.

Defined in packages/core/src/utility/type-of.ts:193

---

core.TypeOf.regex

regex

▸ `Static` **regex**(`value`): value is RegExp

Type-check `value` for `RegExp`.

`Example`

Type-check `null` for `RegExp`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.regex(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is RegExp

Whether `value` is of type `RegExp`.

Defined in packages/core/src/utility/type-of.ts:211

---

core.TypeOf.string

string

▸ `Static` **string**(`value`): value is string

Type-check `value` for `string`.

`Example`

Type-check `null` for `string`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.string(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is string

Whether `value` is of type `string`.

Defined in packages/core/src/utility/type-of.ts:229

---

core.TypeOf.undefined

undefined

▸ `Static` **undefined**(`value`): value is undefined

Type-check `value` for `undefined`.

`Example`

Type-check `null` for `undefined`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.undefined(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is undefined

Whether `value` is of type `undefined`.

Defined in packages/core/src/utility/type-of.ts:247

---

core.TypeOf.url

url

▸ `Static` **url**(`value`): value is URL

Type-check `value` for `URL`.

`Example`

Type-check `null` for `URL`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.url(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is URL

Whether `value` is of type `URL`.

Defined in packages/core/src/utility/type-of.ts:265

---

core.TypeOf.window

window

▸ `Static` **window**(`value`): value is Window

Type-check `value` for `Window`.

`Example`

Type-check `null` for `Window`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.window(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Window

Whether `value` is of type `Window`.

Defined in packages/core/src/utility/type-of.ts:283

---

core.TypeOf.test

test

▸ `Static` `Private` **test**(`type`, `value`): `boolean`

Type-check `value` for `type`.

Parameters

Name	Type	Description
`type`	`string`	Type to check for.
`value`	`unknown`	Value to type-check.

Returns `boolean`

Whether `value` is `type`.

Defined in packages/core/src/utility/type-of.ts:294

---

core.TypeOf.constructor

constructor

• `Private` **new TypeOf**()

Private **constructor** (which should never be called).

`Throws`

TypeError.

Defined in packages/core/src/utility/type-of.ts:303

---

core.assign

assign

▸ **assign**<T, S>(target, ...sources): T & Merge<S[number]>

Assigns (deep copies) the values of all of the enumerable own properties from one or more `source` objects to a `target` object. The last value within the last `source` object takes precedence over any previously encountered values. Returns the `target` object.

Example

Assign nested properties:

```
import { assign } from '@sgrud/core';

assign(
  { one: { one: true }, two: false },
  { one: { key: null } },
  { two: true }
);

// { one: { one: true, key: null }, two: true }
```

Type parameters

Name	Type	Description
T	extends Record<PropertyKey, any>	Target type.
S	extends Record<PropertyKey, any>[]	Source types.

Parameters

Name	Type	Description
target	T	Object to **assign** properties to.
...sources	[...S[]]	Objects from which to deep copy properties.

Returns T & Merge<S[number]>

Assigned object.

Defined in packages/core/src/utility/assign.ts:30

---

core.pluralize

pluralize

▸ **pluralize**(singular): string

Pluralizes words of the English language.

Example

Pluralize 'money':

```
import { pluralize } from '@sgrud/core';

pluralize('money'); // 'money'
```

Example

Pluralize 'thesis':

```
import { pluralize } from '@sgrud/core';

pluralize('thesis'); // 'theses'
```

Parameters

Name	Type	Description
`singular`	`string`	English word in singular form.

Returns `string`

Pluralized form of `singular`.

Defined in packages/core/src/utility/pluralize.ts:23

---

core.provide

provide

• `Const` **provide**: typeof `provide`

Unique symbol used as property key by the Provide type constraint.

Defined in packages/core/src/super/provide.ts:8

---

core.semver

semver

▸ **semver**(`version`, `range`): `boolean`

Best-effort semver matcher. The supplied `version` will be tested against the supplied `range`.

`Example`

Test '1.2.3' against '>2 <1 || ~1.2.*':

```
import { semver } from '@sgrud/core';

semver('1.2.3', '>2 <1 || ~1.2.*'); // true
```

Parameters

Name	Type	Description
`version`	`string`	Tested semantic version string.
`range`	`string`	Range to test the `version` against.

Returns `boolean`

Wether `version` satisfies `range`.

Defined in packages/core/src/kernel/semver.ts:19

Module: data

data

• **data**: `Object`

`@sgrud/data` - The SGRUD Data Model.

The functions and classes found within this module are intended to ease the type safe data handling, i.e., retrieval, mutation and storage, throughout applications building upon the SGRUD client libraries. By

extending the Model class and applying adequate decorators to the contained properties, the resulting extension will, in its static context, provide all necessary means to interact directly with the underlying repository, while the instance context of any class extending the abstract Model base class will inherit methods to observe changes to its instance field values, selectively complement the instance with fields from the backing data storage via type safe graph representations and to delete the respective instance from the data storage.

Defined in packages/data/index.ts:22

---

data.Enum

Enum

• `Abstract` **Enum**: `Object`

Abstract **Enum** helper class. This class is used by the Model to detect enumerations within a Graph, as enumerations (in contrast to plain strings) must not be quoted. This class should never be instantiated manually, but instead is used internally by the enumerate function.

`See`

enumerate

Defined in packages/data/src/model/enum.ts:13

---

data.Enum.constructor

constructor

• `Private` **new Enum**()

Private **constructor** (which should never be called).

`Throws`

TypeError.

Overrides String.constructor

Defined in packages/data/src/model/enum.ts:21

---

data.HasMany

HasMany

▸ **HasMany**<T>(`typeFactory`, `transient?`): <M>(`model`: `M`, `field`: `Field`<`M`>) => `void`

Model field decorator factory. Using this decorator, Models can be enriched with one-to-many associations to other Models. The value for the `typeFactory` argument has to be another Model. By applying this decorator, the decorated field will (depending on the `transient` argument value) be taken into account when serializing or treemapping the Model containing the decorated field.

`Example`

Model with a one-to-many association:

```
import { HasMany, Model } from '@sgrud/data';
import { OwnedModel } from './owned-model';

export class ExampleModel extends Model<ExampleModel> {

  @HasMany(() => OwnedModel)
  public field?: OwnedModel[];

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasOne
- Property

Type parameters

Name	Type	Description
`T`	extends `Type<any, T>`	Field value constructor type.

Parameters

Name	Type	Default value	Description
`typeFactory`	() => `T`	`undefined`	Forward reference to the field value constructor.
`transient`	`boolean`	`false`	Whether the decorated field is transient.

Returns `fn`

Model field decorator.

▸ `<M>(model, field): void`

Type parameters

Name	Type
`M`	extends `Model<any, M>`

Parameters

Name	Type
`model`	`M`
`field`	`Field<M>`

Returns `void`

Defined in packages/data/src/relation/has-many.ts:53

---

data.HasOne

HasOne

▸ **HasOne**`<T>(typeFactory, transient?): <M>(model: M, field: Field<M>) => void`

Model field decorator factory. Using this decorator, Models can be enriched with one-to-one associations to other Models. The value for the `typeFactory` argument has to be another Model. By applying this decorator, the decorated field will (depending on the `transient` argument value) be taken into account when serializing or treemapping the Model containing the decorated field.

Example

Model with a one-to-one association:

```ts
import { HasOne, Model } from '@sgrud/data';
import { OwnedModel } from './owned-model';

export class ExampleModel extends Model<ExampleModel> {

  @HasOne(() => OwnedModel)
  public field?: OwnedModel;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasMany
- Property

Type parameters

Name	Type	Description
`T`	extends `Type<any, T>`	Field value constructor type.

Parameters

Name	Type	Default value	Description
`typeFactory`	`() => T`	`undefined`	Forward reference to the field value constructor.
`transient`	`boolean`	`false`	Whether the decorated field is transient.

Returns `fn`

Model field decorator.

▸ `<M>(model, field): void`

Type parameters

Name	Type
`M`	extends `Model<any, M>`

Parameters

Name	Type
`model`	`M`
`field`	`Field<M>`

Returns `void`

Defined in packages/data/src/relation/has-one.ts:53

---

data.HttpQuerier

HttpQuerier

• **HttpQuerier**: `Object`

HTTP based data Querier, i.e., extension of the abstract Querier base class, allowing Model data queries to be committed via HTTP. To use this class, provide it to the Linker by either extending it, and decorating the extending class with the Target decorator, or by preemptively supplying an instance of this class to the Linker.

`Example`

Provide the **HttpQuerier** to the Linker:

```
import { Linker } from '@sgrud/core';
import { HttpQuerier } from '@sgrud/data';

new Linker<typeof HttpQuerier>([[
  HttpQuerier,
  new HttpQuerier('https://api.example.com')
]]);
```

`See`

- Model
- Querier

Defined in packages/data/src/querier/http.ts:33

---

data.HttpQuerier.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.data.querier.Querier"`

Magic string by which this class is provided.

`See`

provide

Inherited from Querier.[provide]

Defined in packages/data/src/querier/querier.ts:105

---

data.HttpQuerier.commit

commit

▸ **commit**(`operation`, `variables?`): `Observable<any>`

Overridden **commit** method of the Querier base class. When this Model querier is made available via the Linker, this overridden method is called whenever this querier claims the highest *priority* to *commit* an Operation, depending on the Model from which the Operation originates.

`Throws`

Observable of AggregateError.

Parameters

Name	Type	Description
`operation`	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	Querier Operation to be committed.
`variables?`	`Variables`	Variables within the Operation.

Returns `Observable<any>`

Observable of the committed Operation.

Overrides Querier.commit

Defined in packages/data/src/querier/http.ts:101

---

data.HttpQuerier.constructor

constructor

• **new HttpQuerier**(`endpoint?`, `prioritize?`)

Public **constructor** consuming the HTTP `endpoint` Model queries should be fired against, and an dynamic or static `prioritize` value. The `prioritize` value may either be a mapping of Models to corresponding priorities or a static priority for this querier.

Parameters

Name	Type	Default value	Description
`endpoint`	`string`	`undefined`	HTTP endpoint to fire Model queries against.
`prioritize`	`number` | `Map<Type<any>, number>`	`0`	Dynamic or static prioritization.

Overrides Querier.constructor

Defined in packages/data/src/querier/http.ts:60

---

data.HttpQuerier.priority

priority

▸ **priority**(`model`): `number`

Overridden **priority** method of the Querier base class. When an Operation is to be committed, this method is called with the respective `model` constructor and returns the claimed priority to commit this Model.

Parameters

Name	Type	Description
`model`	`Type<any>`	Model to be committed.

Returns `number`

Priority of this implementation.

Overrides Querier.priority

Defined in packages/data/src/querier/http.ts:128

---

data.HttpQuerier.types

types

• `Readonly` **types**: `Set<Type>`

A set containing the the Types this Querier can handle. As HTTP connections are short-lived, this Querier may only handle one-off query Types, namely `'mutation'` and `'query'`.

Overrides Querier.types

Defined in packages/data/src/querier/http.ts:44

---

data.HttpQuerier.endpoint

endpoint

• `Private` `Readonly` **endpoint**: `string`

HTTP endpoint to fire Model queries against.

Default Value

```
new Kernel().endpoint + '/data'
```

Defined in packages/data/src/querier/http.ts:70

---

data.HttpQuerier.prioritize

prioritize

• `Private` `Readonly` **prioritize**: `number` | `Map<Type<any>, number>` = `0`

Dynamic or static prioritization.

Default Value

```
0
```

Defined in packages/data/src/querier/http.ts:77

---

data.Model

Model

• `Abstract` **Model**<`M`>: `Object`

Abstract base class to implement data **Model**s. By extending this abstract base class while providing the enforced symbol property containing the singular name of the resulting data **Model**, type safe data handling, i.e., retrieval, mutation and storage, can easily be achieved. Through the use of the static- and instance-scoped polymorphic `this`, all inherited operations warrant type safety and provide intellisense.

Example

Extend the *Model* base class:

```
import { Model, Property } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  @Property(() => String)
  public field: string?;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

Querier

Type parameters

Name	Type	Description
`M`	extends `Model = any`	Extending *Model* instance type.

Defined in packages/data/src/model/model.ts:20

packages/data/src/model/model.ts:136

packages/data/src/model/model.ts:341

---

data.Model.commit

commit

▸ `Static` **commit**<`T`>(`this`, `operation`, `variables?`): `Observable<any>`

Static **commit** method. Calling this method on a class extending the abstract *Model* base class, while supplying an `operation` and all its embedded `variables`, will dispatch the supplied Operation to the respective *Model* repository through the highest priority Querier or, if no Querier is compatible, throw an error. This method is the central point of origin for all *Model*-related data transferral and is internally called by all other distinct methods of the *Model*.

`Throws`

Observable of ReferenceError.

`Example`

Commit a `query`-type operation:

```
import { ExampleModel } from './example-model';

ExampleModel.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`operation`	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	Operation to **commit**.
`variables?`	`Variables`	Variables within the `operation`.

Returns `Observable<any>`

Observable of the **commit**ment.

Defined in packages/data/src/model/model.ts:379

---

data.Model.deleteAll

deleteAll

▸ `Static` **deleteAll**<`T`>(`this`, `uuids`): `Observable<any>`

Static **deleteAll** method. Calling this method on a class extending the *Model*, while supplying a list of `uuids`, will dispatch the deletion of all *Model* instances identified by these UUIDs to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, bulk-deletions from the respective *Model* repository can be achieved.

`Example`

Drop all model instances by UUIDs:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteAll([
  'b050d63f-cede-46dd-8634-a80d0563ead8',
  'a0164132-cd9b-4859-927e-ba68bc20c0ae',
  'b3fca31e-95cd-453a-93ae-969d3b120712'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`uuids`	`string[]`	UUIDs of *Model* instances to be deleted.

Returns `Observable<any>`

Observable of the deletion.

Defined in packages/data/src/model/model.ts:432

---

data.Model.deleteOne

deleteOne

▸ `Static` **deleteOne**<`T`>(`this`, `uuid`): `Observable<any>`

Static **deleteOne** method. Calling this method on a class extending the *Model*, while supplying an `uuid`, will dispatch the deletion of the *Model* instance identified by this UUID to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, the deletion of a single *Model* instance from the respective *Model* repository can be achieved.

`Example`

Drop one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteOne(
  '18f3aa99-afa5-40f4-90c2-71a2ecc25651'
).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`uuid`	`string`	UUID of the *Model* instance to be deleted.

Returns `Observable<any>`

Observable of the deletion.

Defined in packages/data/src/model/model.ts:468

---

data.Model.findAll

findAll

▸ `Static` **findAll**<T>(`this`, `filter`, `graph`): `Observable<{ result: T[] ; total: number }>`

Static **findAll** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `filter` to match *Model* instances by and a `graph` containing the fields to be included in the result, will dispatch a lookup operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, the bulk-lookup of *Model* instances from the respective *Model* repository can be achieved.

`Example`

Lookup all UUIDs for model instances modified between two dates:

```
import { ExampleModel } from './example-model';

ExampleModel.findAll({
  expression: {
    conjunction: {
      operands: [
        {
          entity: {
            operator: 'GREATER_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-01-01')
          }
        },
        {
          entity: {
            operator: 'LESS_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-12-12')
          }
        }
      ],
      operator: 'AND'
    }
  }
}, [
  'id',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`filter`	`Filter<T>`	Filter to find *Model* instances by.
`graph`	`Graph<T>`	Graph of fields to be included.

Returns `Observable<{ result: T[] ; total: number }>`

Observable of the find operation.

Defined in packages/data/src/model/model.ts:531

---

data.Model.findOne

findOne

▸ `Static` **findOne**`<T>(this, shape, graph): Observable<T>`

Static **findOne** method. Calling this method on a class extending the abstract *Model* base class, while supplying the `shape` to match the *Model* instance by and a `graph` describing the fields to be included in the result, will dispatch the lookup operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, the retrieval of one specific *Model* instance from the respective *Model* repository can be achieved.

`Example`

Lookup one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.findOne({
  id: '2cfe7609-c4d9-4e4f-9a8b-ad72737db48a'
}, [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`shape`	`Shape<T>`	Shape of the *Model* instance to find.
`graph`	`Graph<T>`	Graph of fields to be included.

Returns `Observable<T>`

Observable of the find operation.

Defined in packages/data/src/model/model.ts:583

---

data.Model.saveAll

saveAll

▸ `Static` **saveAll**<T>(`this`, `models`, `graph`): `Observable<T[]>`

Static **saveAll** method. Calling this method on a class extending the abstract *Model* base class, while supplying a list of `models` which to save and a `graph` describing the fields to be included in the result, will dispatch the save operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, bulk-persistance of *Model* instances from the respective *Model* repository can be achieved.

`Example`

Persist multiple *Model*s:

```
import { ExampleModel } from './example-model';

ExampleModel.saveAll([
  new ExampleModel({ field: 'example_1' }),
  new ExampleModel({ field: 'example_2' }),
  new ExampleModel({ field: 'example_3' })
], [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`models`	`T[]`	Array of *Model*s to be saved.
`graph`	`Graph<T>`	Graph of fields to be included.

Returns `Observable<T[]>`

Observable of the save operation.

Defined in packages/data/src/model/model.ts:632

---

data.Model.saveOne

saveOne

▸ `Static` **saveOne**<T>(`this`, `model`, `graph`): `Observable<T>`

Static **saveOne** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `model` which to save and a `graph` describing the fields to be included in the result, will dispatch the save operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, persistance of one specific *Model* instance from the respective *Model* repository can be achieved.

`Example`

Persist a model:

```
import { ExampleModel } from './example-model';

ExampleModel.saveOne(new ExampleModel({ field: 'example' }), [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending *Model* instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
model	T	*Model* which is to be saved.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<T>`

Observable of the save operation.

Defined in packages/data/src/model/model.ts:677

---

data.Model.serialize

serialize

▸ Static **serialize**<T>(this, model, shallow?): undefined | Shape<T>

Static **serialize** method. Calling this method on a class extending the *Model*, while supplying a model which to **serialize** and optionally enabling shallow serialization, will return the Shape of the *Model*, i.e., a plain JSON representation of all *Model* fields, or undefined, if the supplied model does not contain any fields or values. By serializing shallowly, only properties defined on the supplied model are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the serialization of one specific *Model* instance from the respective *Model* repository can be achieved.

Example

Serialize a model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const shape = ExampleModel.serialize(model);
console.log(shape); // { field: 'example' }
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	Static polymorphic this.

Name	Type	Default value	Description
`model`	`T`	`undefined`	*Model* which is to be **serialize**d.
`shallow`	`boolean`	`false`	Whether to **serialize** shallowly.

Returns `undefined | Shape<T>`

Shape of the *Model* or undefined.

Defined in packages/data/src/model/model.ts:721

---

data.Model.treemap

treemap

▸ `Static` **treemap**<T>(`this`, `model`, `shallow?`): `undefined | Graph<T>`

Static **treemap** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `model` which to **treemap** and optionally enabling `shallow` **treemap**ping, will return a Graph describing the fields which are declared and defined on the supplied `model`, or undefined, if the supplied `model` does not contain any fields or values. By **treemap**ping `shallow`ly, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the Graph for one specific *Model* instance from the respective *Model* repository can be retrieved.

`Example`

Treemap a *Model*:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const graph = ExampleModel.treemap(model);
console.log(graph); // ['field']
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
`this`	`Type<T>`	`undefined`	Static polymorphic `this`.
`model`	`T`	`undefined`	*Model* which is to be **treemap**ped.
`shallow`	`boolean`	`false`	Whether to **treemap** shallowly.

Returns `undefined | Graph<T>`

Graph of the *Model* or undefined.

Defined in packages/data/src/model/model.ts:792

---

data.Model.unravel

unravel

▸ `Static` **unravel**<`T`>(`this`, `graph`): `string`

Static **unravel** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `graph` describing the fields which to **unravel**, will return the **unravel**ed Graph as raw string. Through this method, the Graph for one specific *Model* instance from the respective *Model* repository can be **unravel**ed into a raw string. This **unravel**ed Graph can then be consumed by, e.g., the *commit* method.

`Example`

Unravel a Graph:

```
import { ExampleModel } from './example-model';

const unraveled = ExampleModel.unravel([
  'id',
  'modified',
  'field'
]);

console.log(unraveled); // '{id modified field}'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`graph`	`Graph<T>`	Graph which is to be **unravel**ed.

Returns `string`

Unraveled Graph as raw string.

Defined in packages/data/src/model/model.ts:859

---

data.Model.valuate

valuate

▸ `Static` **valuate**<`T`>(`this`, `model`, `field`): `any`

Static **valuate** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `model` and a `field` which to **valuate**, will return the preprocessed value (e.g., primitive representation of JavaScript Dates) of the supplied `field` of the supplied `model`. Through this method, the preprocessed `field` value of one specific *Model* instance from the respective *Model* repository can be retrieved.

`Example`

Valuate a `field`:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ created: new Date(0) });
const value = ExampleModel.valuate(model, 'created');
console.log(value); // '1970-01-01T00:00:00.000+00:00'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`model`	`T`	*Model* which is to be **valuate**d.
`field`	`Field<T>`	Field of the *Model* to be **valuate**d.

Returns `any`

Valuated `field` value.

Defined in packages/data/src/model/model.ts:931

---

data.Model.[hasMany]

[hasMany]

• `Optional` `Readonly` **[hasMany]**: `Record<keyof M, () => unknown>`

Symbol property used by the HasMany decorator.

Defined in packages/data/src/model/model.ts:991

---

data.Model.[hasOne]

[hasOne]

• `Optional` `Readonly` **[hasOne]**: `Record<keyof M, () => unknown>`

Symbol property used by the HasOne decorator.

Defined in packages/data/src/model/model.ts:984

---

data.Model.[observable]

[observable]

• `Readonly` **[observable]**: `() => Subscribable<M>`

Type declaration ▸ (): `Subscribable<M>`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits every mutation this *Model* instance experiences.

`Example`

Subscribe to a *Model* instance:

```
import { from } from 'rxjs';
import { ExampleModel } from './example-model';

const model = new ExampleModel();
from(model).subscribe(console.log);
```

Returns `Subscribable<M>`

Callback to a Subscribable.

Defined in packages/data/src/model/model.ts:1018

---

data.Model.[property]

[property]

• `Optional` `Readonly` **[property]**: `Record<keyof M, () => unknown>`

Symbol property used by the Property decorator.

Defined in packages/data/src/model/model.ts:998

---

data.Model.assign

assign

▸ **assign**<T>(this, ...parts): `Observable<T>`

Instance-scoped **assign** method. Calling this method, while supplying a list of `parts`, will **assign** all supplied `parts` to the *Model* instance. The **assign**ment is implemented as deep merge **assign**ment. Using this method, an existing *Model* instance can easily be mutated while still emitting the mutated *changes*.

`Example`

Assign `parts` to a *Model* instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel();
model.assign({ field: 'example' }).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends `Model<any, T> = M`	Extending *Model* instance type.

Parameters

Name	Type	Description
this	T	Polymorphic `this`.
...parts	Shape<T>[]	Array of parts to **assign**.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1130

---

data.Model.clear

clear

▸ **clear**<T>(this, keys?): `Observable<T>`

Instance-scoped **clear** method. Calling this method on an instance of a class extending the abstract *Model* base class, while optionally supplying a list of `keys` which are to be **clear**ed, will set the value of the properties described by either the supplied `keys` or, if no `keys` were supplied, all enumerable properties of the class extending the abstract *Model* base class to undefined, effectively **clear**ing them.

`Example`

Clear a *Model* instance selectively:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
model.clear(['field']).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending *Model* instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.
keys?	Field<T>[]	Optional array of keys to **clear**.

Returns Observable<T>

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1163

---

data.Model.commit

commit

▸ **commit**<T>(this, operation, variables?, mapping?): Observable<T>

Instance-scoped **commit** method. Internally calls the static *commit* method on the this-context of an instance of a class extending the abstract *Model* base class and furthermore *assign*s the returned data to the *Model* instance the **commit** method was called upon. When supplying a mapping, the returned data will be mutated by the supplied OperatorFunction (otherwise this mapping defaults to identity).

Example

Commit a query-type operation:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel();

model.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
this	T	undefined	Polymorphic this.

Name	Type	Default value	Description
`operation`	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	`undefined`	Operation to **commit**.
`variables?`	`Variables`	`undefined`	Variables within the `operation`.
`mapping`	`OperatorFunction<any, Shape<T>>`	`identity`	OperatorFunction to apply.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1223

---

data.Model.constructor

constructor

• **new Model**<`M`>(...`parts`)

Public **constructor**. The **constructor** of all classes extending the abstract *Model* base class, unless explicitly overridden, behaves analogous to the instance-scoped *assign* method, as it takes all supplied `parts` and assigns them to the instantiated and returned *Model*. The **constructor** furthermore wires some internal functionality, e.g., creates a new *changes* BehaviorSubject which emits every mutation this *Model* instance experiences.

Type parameters

Name	Type
`M`	extends `Model<any, M> = any`

Parameters

Name	Type	Description
`...parts`	`Shape<M>[]`	Array of parts to assign.

Defined in packages/data/src/model/model.ts:1101

---

data.Model.created

created

• `Optional` **created**: `Date`

Transient creation date of this *Model* instance.

Defined in packages/data/src/model/model.ts:1030

---

data.Model.delete

delete

▸ **delete**<T>(this): Observable<T>

Instance-scoped **delete** method. Internally calls the static *deleteOne* method while supplying the UUID of this instance of a class extending the abstract *Model* base class. Calling this method furthermore *clear*s the *Model* instance and completes its deletion by calling complete on the internal *changes* BehaviorSubject of the *Model* instance the **delete** method was called upon.

Example

Drop a *Model* instance by UUID:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({
  id: '3068b30e-82cd-44c5-8912-db13724816fd'
});

model.delete().subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending *Model* instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.

Returns Observable<T>

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1262

---

data.Model.find

find

▸ **find**<T>(this, graph, shape?): Observable<T>

Instance-scoped **find** method. Internally calls the static *findOne* method on the this-context of an instance of a class extending the abstract *Model* base class and then *assign*s the returned data to the *Model* instance the **find** method was called upon.

Example

Find a *Model* instance by UUID:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({
  id: '3068b30e-82cd-44c5-8912-db13724816fd'
});

model.find([
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`T`	Polymorphic `this`.
`graph`	`Graph<T>`	Graph of fields to be included.
`shape`	`Shape<T>`	Shape of the *Model* to find.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1303

---

data.Model.id

id

• `Optional` **id**: `string`

Universally unique identifier of this *Model* instance.

Defined in packages/data/src/model/model.ts:1024

---

data.Model.modified

modified

• `Optional` **modified**: `Date`

Transient modification date of this *Model* instance.

Defined in packages/data/src/model/model.ts:1036

---

data.Model.save

save

▸ **save**<T>(`this`, `graph?`): `Observable<T>`

Instance-scoped **save** method. Internally calls the static *saveOne* method on the `this`-context of an instance of a class extending the abstract *Model* base class and then *assign*s the returned data to the *Model* instance the **save** method was called upon.

`Example`

Persist a *Model* instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });

model.save([
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`T`	Polymorphic `this`.
`graph`	`Graph<T>`	Graph of fields to be included.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1340

---

data.Model.serialize

serialize

▸ **serialize**<T>(`this`, `shallow?`): `undefined` | `Shape<T>`

Instance-scoped **serialize** method. Internally calls the static *serialize* method on the `this`-context of an instance of a class extending the abstract *Model* base class.

`Example`

Serialize a *Model* instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
console.log(model.serialize()); // { field: 'example' }
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
`this`	`T`	`undefined`	Polymorphic `this`.
`shallow`	`boolean`	`false`	Whether to **serialize** shallowly.

Returns `undefined` | `Shape<T>`

Shape of this instance or undefined.

Defined in packages/data/src/model/model.ts:1370

---

data.Model.treemap

treemap

▸ **treemap**<T>(this, shallow?): undefined | Graph<T>

Instance-scoped **treemap** method. Internally calls the static *treemap* method on the this-context of an instance of a class extending the abstract *Model* base class.

Example

Treemap a *Model* instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
console.log(model.treemap()); // ['field']
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
this	T	undefined	Polymorphic this.
shallow	boolean	false	Whether to **treemap** shallowly.

Returns undefined | Graph<T>

Graph of this instance or undefined.

Defined in packages/data/src/model/model.ts:1398

---

data.Model.[toStringTag]

[toStringTag]

• Protected Readonly Abstract **[toStringTag]**: string

Enforced symbol property containing the singular name of this *Model*. The value of this property represents the repository which all instances of this *Model* are considered to belong to. In Detail, the different operations *commit*ted through this *Model* are derived from this singular name (and the corresponding pluralized form).

Example

Provide a valid symbol property:

```
import { Model } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

Defined in packages/data/src/model/model.ts:977

---

data.Model.changes

changes

• Protected Readonly **changes**: BehaviorSubject<M>

BehaviorSubject emitting every time this *Model* instance experiences **changes**.

Defined in packages/data/src/model/model.ts:1044

---

data.Model.entity

entity

• `Protected` `get` **entity**(): `string`

Accessor to the singular name of this *Model*.

Returns `string`

Singular name of this *Model*.

Defined in packages/data/src/model/model.ts:1066

---

data.Model.plural

plural

• `Protected` `get` **plural**(): `string`

Accessor to the **plural**ized name of this *Model*.

Returns `string`

Pluralized name of this *Model*.

Defined in packages/data/src/model/model.ts:1075

---

data.Model.static

static

• `Protected` `Readonly` **static**: `Type<M>`

Type-asserted alias for the **static** *Model* context.

Defined in packages/data/src/model/model.ts:1049

---

data.Model.type

type

• `Protected` `get` **type**(): `string`

Accessor to the raw name of this *Model*.

Returns `string`

Raw name of this *Model*.

Defined in packages/data/src/model/model.ts:1084

---

data.Model

Model

• **Model**: `Object`

Namespace containing types and interfaces used and intended to be used in conjunction with classes extending the abstract Model base class. All the types and interfaces within this namespace are only applicable to classes extending the abstract Model base class, as their generic type argument is always constrained to this abstract base class.

See

Model

Defined in

packages/data/src/model/model.ts:20

packages/data/src/model/model.ts:136

packages/data/src/model/model.ts:341

---

data.Model.Field

Field

T **Field**<`T`>: `string` & `Exclude`<keyof `T`, `Exclude`<keyof `Model`, `"id"` | `"created"` | `"modified"`>>

Type alias for all **Field**s, i.e., own enumerable properties, (excluding internally used ones) of classes extending the abstract Model base class.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending *Model* instance type.

Defined in

packages/data/src/model/model.ts:32

---

data.Model.Filter

Filter

T **Filter**<`T`>: `Params<T>`

Type alias referencing **Filter** Params.

See

Params

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending *Model* instance type.

Defined in

packages/data/src/model/model.ts:45

packages/data/src/model/model.ts:136

---

data.Model.Filter

Filter

• **Filter**: `Object`

Namespace containing types and interfaces to be used when searching through the repositories of classes extending the abstract Model base class. All the interfaces within this namespace are only applicable to classes extending the abstract Model base class, as their generic type argument is always constrained to this abstract base class.

See

Model

Defined in packages/data/src/model/model.ts:45

packages/data/src/model/model.ts:136

Model.Filter.Conjunction

Conjunction

T **Conjunction**: `"AND"` | `"AND_NOT"` | `"OR"` | `"OR_NOT"`

Type alias for a string union type of all possible **Conjunction**s, namely: `'AND'`, `'AND_NOT'`, `'OR'` and `'OR_NOT'`.

Defined in packages/data/src/model/model.ts:142

Model.Filter.Expression

Expression

• **Expression**<`T`>: `Object`

Interface describing the shape of an **Expression** which may be employed through the Params as part of a *findAll* invocation of the Model. **Expression**s can either be the plain shape of an *entity* or compositions of multiple filter expressions, conjunct by one of the Conjunctions.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending *Model* instance type.

Defined in packages/data/src/model/model.ts:175

Filter.Expression.conjunction

conjunction

• `Optional` `Readonly` **conjunction**: `Object`

Conjunction of multiple filter expressions requested data Models are matched against. The *conjunction* sibling parameter has to be undefined when supplying this parameter. By supplying filter expressions, conjunct by specific Conjunction operators, fine-grained filter operations can be compiled.

Type declaration

Name	Type	Description
`operands`	`Expression<T>[]`	List of expressions which are logically combined through an *operator*. These expressions may be nested and can be used to construct complex composite filter operations.
`operator?`	`Conjunction`	Conjunction **operator** used to logically combine all supplied *operands*.

Defined in packages/data/src/model/model.ts:187

Filter.Expression.entity

entity

• `Optional` `Readonly` **entity**: `Object`

Shape the requested data Models are matched against. Supplying this parameter requires the *conjunction* sibling parameter to be undefined. By specifying the **entity** shape to match data Models against, simple filter operations can be compiled.

Type declaration

Name	Type	Description
`operator?`	`Operator`	Operator to use for matching.
`path`	`Path<T, []>`	Property **path** from within the data Model which to match against. The value which will be matched against has to be supplied through the *value* property.
`value`	`unknown`	Property **value** to match data Models against. The property path of this value has to be supplied through the *path* property.

Defined in packages/data/src/model/model.ts:214

Model.Filter.Operator

Operator

T **Operator**: `"EQUAL"` | `"GREATER_OR_EQUAL"` | `"GREATER_THAN"` | `"LESS_OR_EQUAL"` | `"LESS_THAN"` | `"LIKE"` | `"NOT_EQUAL"`

Type alias for a string union type of all possible **Operator**s, namely: `'EQUAL'`, `'NOT_EQUAL'`, `'LIKE'`, `'GREATER_THAN'`, `'GREATER_OR_EQUAL'`, `'LESS_THAN'` and `'LESS_OR_EQUAL'`.

Defined in packages/data/src/model/model.ts:153

Model.Filter.Params

Params

• **Params**<T>: `Object`

Interface describing the **Params** of, e.g., the Model *findAll* method. This is the most relevant interface within this namespace (and is therefore also referenced by the Filter type alias), as it describes the input **Params** of any selective data retrieval.

See

Model

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending *Model* instance type.

Defined in

packages/data/src/model/model.ts:257

Filter.Params.dir

dir

• `Optional` `Readonly` **dir**: `"desc"` | `"asc"`

Desired sorting **dir**ection of the requested data Models. To specify which field the results should be sorted by, the *sort* property must be supplied.

Defined in

packages/data/src/model/model.ts:266

Filter.Params.expression

expression

• `Optional` `Readonly` **expression**: `Expression`<T>

Expression to evaluate results against. This **expression** may be a simple matching or more complex, conjunct and nested **expression**s.

Defined in

packages/data/src/model/model.ts:272

Filter.Params.page

page

• `Optional` `Readonly` **page**: `number`

Page number, i.e., offset within the list of all results for a data Model request. This property should be used together with the page *size* property.

Defined in

packages/data/src/model/model.ts:281

Filter.Params.search

search

• `Optional` `Readonly` **search**: `string`

Free-text **search** field. This field overrides all *expression*s, as such that if this field contains a value, all *expression*s are ignored and only this free-text **search** filter is applied.

Defined in packages/data/src/model/model.ts:288

Filter.Params.size

size

• `Optional` `Readonly` **size**: `number`

Page **size**, i.e., number of results which should be included within the response to a data Model request. This property should be used together with the *page* offset property.

Defined in packages/data/src/model/model.ts:297

Filter.Params.sort

sort

• `Optional` `Readonly` **sort**: `Path<T, []>`

Property path used to determine the value which to **sort** the requested data Models by. This property should be used together with the sorting *dir*ection property.

Defined in packages/data/src/model/model.ts:306

---

data.Model.Graph

Graph

T **Graph**<T>: { [K in Field<T>]?: Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> | Model<infer I>[] ? Record<K, Graph<I> | Function> : K }[Field<T>][]

Mapped type to compile strongly typed **Graph**s of classes extending the abstract Model base class, while providing intellisense.

Type parameters

Name	Type	Description
T	extends Model	Extending *Model* instance type.

Defined in packages/data/src/model/model.ts:55

---

data.Model.Path

Path

T **Path**<T, S>: { [K in Field<T>]: S extends Object ? never : Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> | Model<infer I>[] ? `${K}.${Path<I, [...S, string]>}` : K }[Field<T>]

Mapped type to compile strongly typed property **Path**s of classes extending the abstract Model base class, while providing intellisense.

Type parameters

Name	Type	Description
T	extends Model	Extending *Model* instance type.
S	extends string[] = []	String array type.

Defined in packages/data/src/model/model.ts:73

---

data.Model.Shape

Shape

T **Shape**<T>: { [K in Field<T>]?: Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> ? Shape<I> : Required<T>[K] extends Model<infer I>[] ? Shape<I>[] : Required<T>[K] }

Mapped type to compile strongly typed **Shape**s of classes extending the abstract Model base class, while providing intellisense.

Type parameters

Name	Type	Description
T	extends Model	Extending *Model* instance type.

Defined in packages/data/src/model/model.ts:92

---

data.Model.Type

Type

• **Type**<T>: Object

Interface describing the **Type**, i.e., static constructable context, of classes extending the abstract Model base class.

Type parameters

Name	Type	Description
T	extends Model	Extending *Model* instance type.

Defined in packages/data/src/model/model.ts:111

Model.Type.commit

commit

▸ **commit**<T>(this, operation, variables?): Observable<any>

Static **commit** method. Calling this method on a class extending the abstract *Model* base class, while supplying an operation and all its embedded variables, will dispatch the supplied Operation to the respective *Model* repository through the highest priority Querier or, if no Querier is compatible, throw an error. This method is the central point of origin for all *Model*-related data transferral and is internally called by all other distinct methods of the *Model*.

Throws

Observable of ReferenceError.

Example

Commit a query-type operation:

```
import { ExampleModel } from './example-model';

ExampleModel.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
```

```
}`, {
  variable: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`operation`	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	Operation to **commit**.
`variables?`	`Variables`	Variables within the `operation`.

Returns `Observable<any>`

Observable of the **commit**ment.

Inherited from Required.commit

Defined in packages/data/src/model/model.ts:379

Model.Type.constructor

constructor

• **new Type**(...`args`)

Overridden and concretized constructor signature.

Parameters

Name	Type	Description
`...args`	`Shape<Model<any>>[]`	Class constructor rest parameter.

Inherited from Required<typeof Model>.constructor

Defined in packages/data/src/model/model.ts:118

Model.Type.deleteAll

deleteAll

▸ **deleteAll**<T>(`this`, `uuids`): `Observable<any>`

Static **deleteAll** method. Calling this method on a class extending the *Model*, while supplying a list of `uuids`, will dispatch the deletion of all *Model* instances identified by these UUIDs to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, bulk-deletions from the respective *Model* repository can be achieved.

`Example`

Drop all model instances by UUIDs:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteAll([
  'b050d63f-cede-46dd-8634-a80d0563ead8',
  'a0164132-cd9b-4859-927e-ba68bc20c0ae',
  'b3fca31e-95cd-453a-93ae-969d3b120712'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending *Model* instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
uuids	string[]	UUIDs of *Model* instances to be deleted.

Returns Observable<any>

Observable of the deletion.

Inherited from Required.deleteAll

Defined in packages/data/src/model/model.ts:432

Model.Type.deleteOne

deleteOne

▸ **deleteOne**<T>(this, uuid): Observable<any>

Static **deleteOne** method. Calling this method on a class extending the *Model*, while supplying an uuid, will dispatch the deletion of the *Model* instance identified by this UUID to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, the deletion of a single *Model* instance from the respective *Model* repository can be achieved.

Example

Drop one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteOne(
  '18f3aa99-afa5-40f4-90c2-71a2ecc25651'
).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending *Model* instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.

Name	Type	Description
`uuid`	`string`	UUID of the *Model* instance to be deleted.

Returns `Observable<any>`

Observable of the deletion.

Inherited from Required.deleteOne

Defined in packages/data/src/model/model.ts:468

Model.Type.findAll

findAll

▸ **findAll**<T>(`this`, `filter`, `graph`): `Observable<{ result: T[] ; total: number }>`

Static **findAll** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `filter` to match *Model* instances by and a `graph` containing the fields to be included in the result, will dispatch a lookup operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, the bulk-lookup of *Model* instances from the respective *Model* repository can be achieved.

`Example`

Lookup all UUIDs for model instances modified between two dates:

```
import { ExampleModel } from './example-model';

ExampleModel.findAll({
  expression: {
    conjunction: {
      operands: [
        {
          entity: {
            operator: 'GREATER_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-01-01')
          }
        },
        {
          entity: {
            operator: 'LESS_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-12-12')
          }
        }
      ],
      operator: 'AND'
    }
  }
}, [
  'id',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`filter`	`Filter<T>`	Filter to find *Model* instances by.
`graph`	`Graph<T>`	Graph of fields to be included.

Returns `Observable<{ result: T[] ; total: number }>`

Observable of the find operation.

Inherited from Required.findAll

Defined in packages/data/src/model/model.ts:531

Model.Type.findOne

findOne

▸ **findOne**<T>(this, shape, graph): `Observable<T>`

Static **findOne** method. Calling this method on a class extending the abstract *Model* base class, while supplying the `shape` to match the *Model* instance by and a `graph` describing the fields to be included in the result, will dispatch the lookup operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, the retrieval of one specific *Model* instance from the respective *Model* repository can be achieved.

`Example`

Lookup one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.findOne({
  id: '2cfe7609-c4d9-4e4f-9a8b-ad72737db48a'
}, [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`shape`	`Shape<T>`	Shape of the *Model* instance to find.
`graph`	`Graph<T>`	Graph of fields to be included.

Returns `Observable<T>`

Observable of the find operation.

Inherited from Required.findOne

Defined in packages/data/src/model/model.ts:583

Model.Type.prototype

prototype

• **prototype**: `Model<any>`

Inherited from Required.prototype

Model.Type.saveAll

saveAll

▸ **saveAll**<T>(this, models, graph): `Observable<T[]>`

Static **saveAll** method. Calling this method on a class extending the abstract *Model* base class, while supplying a list of `models` which to save and a `graph` describing the fields to be included in the result, will dispatch the save operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, bulk-persistance of *Model* instances from the respective *Model* repository can be achieved.

`Example`

Persist multiple *Model*s:

```
import { ExampleModel } from './example-model';

ExampleModel.saveAll([
  new ExampleModel({ field: 'example_1' }),
  new ExampleModel({ field: 'example_2' }),
  new ExampleModel({ field: 'example_3' })
], [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
this	`Type<T>`	Static polymorphic `this`.
models	`T[]`	Array of *Model*s to be saved.
graph	`Graph<T>`	Graph of fields to be included.

Returns `Observable<T[]>`

Observable of the save operation.

Inherited from Required.saveAll

Defined in packages/data/src/model/model.ts:632

Model.Type.saveOne

saveOne

▸ **saveOne**<T>(this, model, graph): Observable<T>

Static **saveOne** method. Calling this method on a class extending the abstract *Model* base class, while supplying a model which to save and a graph describing the fields to be included in the result, will dispatch the save operation to the respective *Model* repository by internally calling the *commit* operation with suitable arguments. Through this method, persistance of one specific *Model* instance from the respective *Model* repository can be achieved.

Example

Persist a model:

```
import { ExampleModel } from './example-model';

ExampleModel.saveOne(new ExampleModel({ field: 'example' }), [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending *Model* instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
model	T	*Model* which is to be saved.
graph	Graph<T>	Graph of fields to be included.

Returns Observable<T>

Observable of the save operation.

Inherited from Required.saveOne

Defined in packages/data/src/model/model.ts:677

Model.Type.serialize

serialize

▸ **serialize**<T>(this, model, shallow?): undefined | Shape<T>

Static **serialize** method. Calling this method on a class extending the *Model*, while supplying a model which to **serialize** and optionally enabling shallow serialization, will return the Shape of the *Model*, i.e., a plain JSON representation of all *Model* fields, or undefined, if the supplied model does not contain any fields or values. By serializing shallowly, only properties defined on the supplied model are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the serialization of one specific *Model* instance from the respective *Model* repository can be achieved.

Example

Serialize a model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const shape = ExampleModel.serialize(model);
console.log(shape); // { field: 'example' }
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
`this`	`Type<T>`	`undefined`	Static polymorphic `this`.
`model`	`T`	`undefined`	*Model* which is to be **serialize**d.
`shallow`	`boolean`	`false`	Whether to **serialize** shallowly.

Returns `undefined | Shape<T>`

Shape of the *Model* or undefined.

Inherited from Required.serialize

Defined in packages/data/src/model/model.ts:721

Model.Type.treemap

treemap

▸ **treemap**<T>(`this`, `model`, `shallow?`): `undefined | Graph<T>`

Static **treemap** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `model` which to **treemap** and optionally enabling `shallow` **treemap**ping, will return a Graph describing the fields which are declared and defined on the supplied `model`, or undefined, if the supplied `model` does not contain any fields or values. By **treemap**ping `shallow`ly, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the Graph for one specific *Model* instance from the respective *Model* repository can be retrieved.

`Example`

Treemap a *Model*:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const graph = ExampleModel.treemap(model);
console.log(graph); // ['field']
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Default value	Description
`this`	`Type<T>`	`undefined`	Static polymorphic `this`.
`model`	`T`	`undefined`	*Model* which is to be **treemap**ped.

Name	Type	Default value	Description
`shallow`	`boolean`	`false`	Whether to **treemap** shallowly.

Returns `undefined | Graph<T>`

Graph of the *Model* or undefined.

Inherited from Required.treemap

Defined in packages/data/src/model/model.ts:792

Model.Type.unravel

unravel

▸ **unravel**<T>(this, graph): `string`

Static **unravel** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `graph` describing the fields which to **unravel**, will return the **unravel**ed Graph as raw string. Through this method, the Graph for one specific *Model* instance from the respective *Model* repository can be **unravel**ed into a raw string. This **unravel**ed Graph can then be consumed by, e.g., the *commit* method.

`Example`

Unravel a Graph:

```
import { ExampleModel } from './example-model';

const unraveled = ExampleModel.unravel([
  'id',
  'modified',
  'field'
]);

console.log(unraveled); // '{id modified field}'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending *Model* instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic `this`.
`graph`	`Graph<T>`	Graph which is to be **unravel**ed.

Returns `string`

Unraveled Graph as raw string.

Inherited from Required.unravel

Defined in packages/data/src/model/model.ts:859

Model.Type.valuate

valuate

▸ **valuate**<T>(this, model, field): any

Static **valuate** method. Calling this method on a class extending the abstract *Model* base class, while supplying a `model` and a `field` which to **valuate**, will return the preprocessed value (e.g., primitive representation of JavaScript Dates) of the supplied `field` of the supplied `model`. Through this method, the preprocessed `field` value of one specific *Model* instance from the respective *Model* repository can be retrieved.

Example

Valuate a `field`:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ created: new Date(0) });
const value = ExampleModel.valuate(model, 'created');
console.log(value); // '1970-01-01T00:00:00.000+00:00'
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending *Model* instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
model	T	*Model* which is to be **valuate**d.
field	Field<T>	Field of the *Model* to be **valuate**d.

Returns any

Valuated `field` value.

Inherited from Required.valuate

Defined in packages/data/src/model/model.ts:931

---

data.Property

Property

T **Property**: Type<any> | typeof Boolean | typeof Date | typeof Number | typeof String

Type alias for a union type of all primitive constructors which may be used as `typeFactory` argument for the Property decorator.

See

Property

Defined in packages/data/src/relation/property.ts:70

packages/data/src/relation/property.ts:12

---

data.Property

Property

▸ **Property**<T>(typeFactory, transient?): <M>(model: M, field: Field<M>) => void

Model field decorator factory. Using this decorator, Models can be enriched with primitive fields. The compatible primitives are the subset of primitives JavaScript shares with JSON, i.e., *Boolean*, *Date* (serialized), *Number* and *String*. *Objects* cannot be uses as a `typeFactory` argument value, as Model fields containing objects should be declared by the HasOne and HasMany Model field decorators. By employing this decorator, the decorated field will (depending on the `transient` argument value) be taken into account when serializing or treemapping the Model containing the decorated field.

Example

Model with a primitive field:

```
import { Model, Property } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  @Property(() => String)
  public field?: string;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasOne
- HasMany

Type parameters

Name	Type	Description
T	extends Property	Field value constructor type.

Parameters

Name	Type	Default value	Description
typeFactory	() => T	undefined	Forward reference to the field value constructor.
transient	boolean	false	Whether the decorated field is transient.

Returns fn

Model field decorator.

▸ <M>(model, field): void

Type parameters

Name	Type
M	extends Model<any, M>

Parameters

Name	Type
model	M
field	Field<M>

Returns `void`

Defined in packages/data/src/relation/property.ts:70

---

data.Querier

Querier

• `Abstract` **Querier**: `Object`

Abstract **Querier** base class to implement Model data queriers. By extending this abstract base class and providing the extending class to the Linker, e.g., by Targeting it, the respective classes *priority* method will be called whenever the Model *commits* data and, if this class claims the highest priority, its *commit* method will be called.

`Decorator`

Provide

`Example`

Simple **Querier** stub:

```
import type { Model, Querier } from '@sgrud/data';
import type { Observable } from 'rxjs';
import { Provider, Target } from '@sgrud/core';

@Target<typeof ExampleQuerier>()
export class ExampleQuerier
  extends Provider<typeof Querier>('sgrud.data.querier.Querier') {

  public override readonly types: Set<Querier.Type> = new Set<Querier.Type>([
    'query'
  ]);

  public override commit(
    operation: Querier.Operation,
    variables: Querier.Variables
  ): Observable<any> {
    throw new Error('Stub!');
  }

  public override priority(model: Model.Type<any>): number {
    return 0;
  }

}
```

`See`

Model

Defined in packages/data/src/querier/querier.ts:15

packages/data/src/querier/querier.ts:96

---

data.Querier.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.data.querier.Querier"`

Magic string by which this class is provided.

`See`

provide

Defined in packages/data/src/querier/querier.ts:105

---

data.Querier.commit

commit

▸ `Abstract` **commit**(`operation`, `variables?`): `Observable<any>`

The overridden **commit** method of Targeted queriers is called by the Model to **commit** operations. The invocation arguments are the `operation`, unraveled into a string, and all `variables` embedded within this operation. The extending class has to serialize the Variables and transfer the operation. It's the callers responsibility to unravel the Operation prior to invoking this method, and to deserialize and (error) handle whatever response is received.

Parameters

Name	Type	Description
`operation`	'mutation ${string}' | 'query ${string}' | 'subscription ${string}'	Querier Operation to be committed.
`variables?`	`Variables`	Variables within the Operation.

Returns `Observable<any>`

Observable of the committed Operation.

Defined in packages/data/src/querier/querier.ts:135

---

data.Querier.constructor

constructor

• **new Querier**()

---

data.Querier.priority

priority

▸ `Abstract` **priority**(`model`): `number`

Whenever the *commit* method of the Model is invoked, all Targeted and compatible queriers, i.e., implementations of the this class capable of handling the specific Type of the to be committed Operation, will be asked to prioritize themselves regarding the respective Model. The querier claiming the highest **priority** will be chosen and its *commit* method called.

Parameters

Name	Type	Description
`model`	`Type<any>`	Model to be committed.

Returns `number`

Priority of this implementation.

Defined in packages/data/src/querier/querier.ts:156

---

data.Querier.types

types

• `Readonly` `Abstract` **types**: `Set<Type>`

A set containing all Types of queries this class can handle. May contain none to all of `'mutation'`, `'query'` and `'subscription'`.

Defined in

packages/data/src/querier/querier.ts:114

---

data.Querier

Querier

• **Querier**: `Object`

Namespace containing types and interfaces used and intended to be used in conjunction with the abstract Querier base class and in context of the Model data handling.

See

Querier

Defined in

packages/data/src/querier/querier.ts:15

packages/data/src/querier/querier.ts:96

---

data.Querier.Operation

Operation

T **Operation**: `${Type} ${string}`

String literal helper type. Enforces any assigned string to conform to the standard form of an operation: A string, starting with the Type, followed by one whitespace and the operation content.

Defined in

packages/data/src/querier/querier.ts:35

---

data.Querier.Type

Type

T **Type**: `"mutation"` | `"query"` | `"subscription"`

Type alias for a string union type of all known Operation types: `'mutation'`, `'query'` and `'subscription'`.

Defined in

packages/data/src/querier/querier.ts:23

---

data.Querier.Variables

Variables

• **Variables**: `Object`

Interface describing the shape of variables which may be embedded within Operations. Variables are a simple key-value map, which can be deeply nested.

Defined in

packages/data/src/querier/querier.ts:44

---

data.enumerate

enumerate

▸ **enumerate**<T>(`enumerator`): `T`

Enumerate helper function. Enumerations are special objects and all used TypeScript enums have to be looped through this helper function before they can be utilized in conjunction with the Model.

`Example`

Enumerate a TypeScript enumeration:

```
import { enumerate } from '@sgrud/data';

enum Enumeration {
  One = 'ONE',
  Two = 'TWO'
}

export type ExampleEnum = Enumeration;
export const ExampleEnum = enumerate(Enumeration);
```

`See`

Model

Type parameters

Name	Type	Description
`T`	extends `object`	Enumeration type.

Parameters

Name	Type	Description
`enumerator`	`T`	TypeScript enumeration.

Returns `T`

Processed enumeration.

Defined in packages/data/src/model/enum.ts:55

---

data.hasMany

hasMany

• `Const` **hasMany**: typeof `hasMany`

Unique symbol used as property key by the HasMany decorator to register decorated Model fields for further computation, e.g., serialization, treemapping etc.

`See`

HasMany

Defined in packages/data/src/relation/has-many.ts:14

---

data.hasOne

hasOne

• `Const` **hasOne**: typeof `hasOne`

Unique symbol used as property key by the HasOne decorator to register decorated Model fields for further computation, e.g., serialization, treemapping etc.

See

HasOne

Defined in

packages/data/src/relation/has-one.ts:14

---

data.property

property

• `Const` **property**: typeof `property`

Unique symbol used as property key by the Property decorator to register decorated Model fields for further computation, e.g., serialization, treemapping etc.

See

Property

Defined in

packages/data/src/relation/property.ts:29

Module: shell

shell

• **shell**: `Object`

`@sgrud/shell` - The SGRUD Web UI Shell.

The functions and classes found within this module are intended to ease the implementation of Component-based frontends by providing JSX runtime bindings for the incremental-dom library and a Router targeted at routing through Components based upon the SGRUD client libraries, but not limited to those. Furthermore, complex routing strategies and actions may be implemented through the interceptor-like Router-Task pattern.

Defined in

packages/shell/index.ts:21

---

shell.Attribute

Attribute

▸ **Attribute**(`name?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Component prototype property decorator factory. Applying the **Attribute** decorator to a property of a Component binds the decorated property to the corresponding attribute of the respective Component. This implies that the attribute `name` is added to the *observedAttributes* array of the Component and the decorated property is replaced with a getter and setter deferring those operations to the attribute. If no `name` supplied, the name of the decorated property will be used instead. Further, if both, a parameter initializer and an initial attribute value are supplied, the attribute value takes precedence.

Example

Decorate a property:

```
import { Attribute, Component } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
```

```
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Attribute()
  public field?: string;

  public get template(): JSX.Element {
    return <span>Attribute value: {this.field}</span>;
  }

}
```

See

Component

Parameters

Name	Type	Description
`name?`	`string`	Component attribute name.

Returns `fn`

Component prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/component/attribute.ts:46

---

shell.Component

Component

▸ **Component**<`S`, `R`>(`selector`, `inherits?`): <`T`>(`constructor`: `T`) => `T`

Class decorator factory. Registers the decorated class as **Component** through the customElements registry. Registered **Component**s can be used in conjunction with the Attribute and Reference prototype property decorators which will trigger the **Component** to re-render, when one of the *observedAttributes* or *observedReferences* changes. While any **Component** which is registered by this decorator is enriched with basic rendering functionality, any implemented method will cancel out its `super` logic.

Example

Register a component:

```
import { Component } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {
```

```
  public readonly styles: string[] = [`
    span {
      font-style: italic;
    }
  `];

  public get template(): JSX.Element {
    return <span>Example component</span>;
  }

}
```

See

- Attribute
- Reference

Type parameters

Name	Type	Description
S	extends `CustomElementTagName`	**Component** tag type.
R	extends `HTMLElementTagName`	-

Parameters

Name	Type	Description
`selector`	S	**Component** tag name.
`inherits?`	R	Extended tag name.

Returns `fn`

Class decorator.

▸ <T>(`constructor`): T

Type parameters

Name	Type
T	extends () => `Component` & `HTMLElementTagNameMap[S]` & `HTMLElementTagNameMap[R]`

Parameters

Name	Type
`constructor`	T

Returns `T`

Defined in packages/shell/src/component/component.ts:157

---

shell.Component

Component

• **Component**: `Object`

Interface describing the shape of a **Component**. Mostly adheres to the WebComponents specification while providing rendering and change detection capabilities.

Defined in packages/shell/src/component/component.ts:157

packages/shell/src/component/component.ts:19

---

shell.Component.adoptedCallback

adoptedCallback

▸ `Optional` **adoptedCallback**(): `void`

Called when the *Component* is moved between documents.

Returns `void`

Defined in packages/shell/src/component/component.ts:61

---

shell.Component.attributeChangedCallback

attributeChangedCallback

▸ `Optional` **attributeChangedCallback**(`name`, `prev?`, `next?`): `void`

Called when one of the *Component*'s observed Attributes is added, removed or changed. Which *Component* attributes are observed depends on the contents of the *observedAttributes* array.

Parameters

Name	Type	Description
`name`	`string`	Attribute name.
`prev?`	`string`	Previous value.
`next?`	`string`	Next value.

Returns `void`

Defined in packages/shell/src/component/component.ts:74

---

shell.Component.connectedCallback

connectedCallback

▸ `Optional` **connectedCallback**(): `void`

Called when the *Component* is appended to or moved within the dom.

Returns `void`

Defined in packages/shell/src/component/component.ts:79

---

shell.Component.constructor

constructor

• **constructor**: `Object`

Inherited from HTMLElement.constructor

---

shell.Component.disconnectedCallback

disconnectedCallback

▸ `Optional` **disconnectedCallback**(): `void`

Called when the *Component* is removed from the dom.

Returns `void`

Defined in packages/shell/src/component/component.ts:84

---

shell.Component.observedAttributes

observedAttributes

• `Optional` `Readonly` **observedAttributes**: `string`[]

Array of Attribute names, which should be observed for changes, which will trigger the *attributeChanged-Callback*.

Defined in packages/shell/src/component/component.ts:27

---

shell.Component.observedReferences

observedReferences

• `Optional` `Readonly` **observedReferences**: `Record<Key, keyof HTMLElementEventMap[]>`

Mapping of References to observed events, which, when emitted by the referenced node, trigger the *referenceChangedCallback*.

Defined in packages/shell/src/component/component.ts:35

---

shell.Component.readyState

readyState

• `Optional` `Readonly` **readyState**: `boolean`

Internal readiness indication. Initially resolves to `undefined` and will mirror the *isConnected* state, when ready.

Defined in packages/shell/src/component/component.ts:41

---

shell.Component.referenceChangedCallback

referenceChangedCallback

▸ `Optional` **referenceChangedCallback**(`name`, `node`, `event`): `void`

Called when one of the *Component*'s Referenced and observed nodes emits an event. Which Referenced nodes are observed for which events depends on the contents of the *observedReferences* mapping.

Parameters

Name	Type	Description
`name`	`string`	Reference name.
`node`	`Node`	-
`event`	`Event`	Emitted event.

Returns `void`

Defined in packages/shell/src/component/component.ts:96

---

shell.Component.renderComponent

renderComponent

▸ `Optional` **renderComponent**(): `void`

Called when the *Component* has changed and should be (re-)rendered.

Returns `void`

Defined in packages/shell/src/component/component.ts:103

---

shell.Component.styles

styles

• `Optional` `Readonly` **styles**: `string`[]

Array of CSS **styles** in string form, which should be included within the shadow dom of the *Component*.

Defined in packages/shell/src/component/component.ts:47

---

shell.Component.template

template

• `Optional` `Readonly` **template**: `Element`

JSX representation of the *Component* **template**. If no template is supplied, an HTMLSlotElement will be rendered instead.

Defined in packages/shell/src/component/component.ts:56

---

shell.CustomElementTagName

CustomElementTagName

T **CustomElementTagName**: `Extract`<keyof `HTMLElementTagNameMap`, '${string}-${string}'>

Global string literal helper type. Enforces any assigned string to be a `keyof HTMLElementTagNameMap`, while excluding built-in tag names, i.e., extracting all `${string}-${string}` keys of `HTMLElementTagNameMap`.

`Example`

A valid **CustomElementTagName**:

```
const tagName: CustomElementTagName = 'example-component';
```

Defined in packages/shell/src/component/runtime.ts:17

---

shell.HTMLElementTagName

HTMLElementTagName

T **HTMLElementTagName**: `Exclude`<keyof `HTMLElementTagNameMap`, '`${string}-${string}`'>

Global string literal helper type. Enforces any assigned string to be a `keyof HTMLElementTagNameMap`, while excluding custom element tag names, i.e., all `${string}-${string}` keys of `HTMLElementTagNameMap`.

`Example`

A valid **HTMLElementTagName**:

```
const tagName: HTMLElementTagName = 'div';
```

Defined in

packages/shell/src/component/runtime.ts:31

---

shell.JSX

JSX

• **JSX**: `Object`

Intrinsic JSX namespace.

Defined in

packages/shell/src/component/runtime.ts:39

---

shell.JSX.Element

Element

T **Element**: () => `Node`[]

Intrinsic JSX **element** type helper representing an array of bound incremental-dom calls.

Defined in

packages/shell/src/component/runtime.ts:48

---

shell.JSX.IntrinsicElements

IntrinsicElements

T **IntrinsicElements**: { [K in keyof HTMLElementTagNameMap]: Partial<HTMLElementTagNameMap[K]> & Object }

Intrinsic list of known JSX elements, comprised of the global `HTMLElementTagNameMap`.

Defined in

packages/shell/src/component/runtime.ts:56

---

shell.JSX.Key

Key

T **Key**: `string` | `number`

Element reference **Key** type helper. Enforces any assigned value to to be a incremental-dom-compatible **Key** type.

Defined in

packages/shell/src/component/runtime.ts:80

---

shell.Reference

Reference

▸ **Reference**(`reference`, `observe?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Component prototype property decorator factory. Applying the **Reference** decorator to a property of a registered Component while supplying the `reference` key and, optionally, an array of events to `observe`, will replace the decorated property with a getter returning the referenced node, once rendered. If an array of events is supplied, whenever one of those events is emitted by the referenced node, the *referenceChangedCallback* of the respective Component is called with the `reference` key, the referenced node and the emitted event.

`Example`

Reference a node:

```
import { Component, Reference } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Reference('example-key')
  private readonly span?: HTMLSpanElement;

  public get template(): JSX.Element {
    return <span key="example-key"></span>;
  }

}
```

`See`

Component

Parameters

Name	Type	Description
`reference`	`Key`	Element reference.
`observe?`	keyof `HTMLElementEventMap`[]	Events to observe.

Returns `fn`

Component prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/component/reference.ts:49

---

shell.Route

Route

▸ **Route**<S>(config): <T>(constructor: T) => void

Class decorator factory. Applying the **Route** decorator to a custom element will associate the supplied Route `config` to the decorated element constructor. Further, the configured children are iterated and every child that is a custom element itself will be replaced by its respective Route. Finally, the processed `config` for the decorated element is associated to the element constructor and added to the Router.

`Example`

Associate a Route `config` to a element:

```
import { Component, Route } from '@sgrud/shell';
import { ChildComponent } from './child-component';

@Route({
  path: 'example',
  children: [
    ChildComponent
  ]
})
@Component('example-element')
export class ExampleComponent extends HTMLElement implements Component { }
```

`See`

Router

Type parameters

Name	Type	Description
S	extends `string`	Route path string type.

Parameters

Name	Type	Description
config	Assign<{ children?: (Route<string> | CustomElementConstructor & { [route]?: Route<string> })[]; slots?: Record<string, CustomElementConstructor | CustomElementTagName> }, Omit<Route<S>, "component">> & { parent?: Route<string> | CustomElementConstructor & { [route]?: Route<string> } }	Route config for this element.

Returns `fn`

Class decorator.

▸ <T>(constructor): void

Type parameters

Name	Type
T	extends CustomElementConstructor & { [route]?: Route<S> }

Parameters

Name	Type
`constructor`	`T`

Returns `void`

Defined in packages/shell/src/router/route.ts:99

---

shell.Route

Route

• **Route**<`S`>: `Object`

Interface describing the shape of a **Route**. A **Route** must consist of at least a *path* and may declare a *component*, which will be rendered when the **Route** is navigated to, as well as *slots* and elements which will be slotted within those. Furthermore a **Route** may also specify *children*.

`Example`

Define a **Route**:

```
import type { Route } from '@sgrud/shell';

const route: Route = {
  path: '',
  component: 'example-element',
  children: [
    {
      path: 'child',
      component: 'child-element'
    }
  ]
};
```

`See`

Router

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	**Route** path string type.

Defined in packages/shell/src/router/route.ts:99

packages/shell/src/router/route.ts:34

---

shell.Route.children

children

• `Optional` `Readonly` **children**: `Route`<`string`>[]

Optional array of **children** for this route.

Defined in packages/shell/src/router/route.ts:39

---

shell.Route.component

component

• `Optional` `Readonly` **component**: `CustomElementTagName`

Optional route **component**.

Defined in packages/shell/src/router/route.ts:44

---

shell.Route.constructor

constructor

• **constructor**: `Object`

---

shell.Route.path

path

• `Readonly` **path**: `S`

Required route **path**.

Defined in packages/shell/src/router/route.ts:49

---

shell.Route.slots

slots

• `Optional` `Readonly` **slots**: `Record<string, CustomElementTagName>`

Optional mapping of **slots** to their elements.

Defined in packages/shell/src/router/route.ts:54

---

shell.Router

Router

• **Router**: `Object`

Targeted Singleton Router class extending the built-in *Set*. This Singleton class provides routing and rendering capabilities. Routing is primarily realized by maintaining (inheriting) a *Set* of Routes and (recursively) *match*ing paths against those Routes, when instructed so by calling *navigate*. When a matching Segment is found, the corresponding Components are rendered by the *handle* method (which is part of the implemented Task contract).

`Decorator`

Target

`Decorator`

Singleton

Defined in packages/shell/src/router/router.ts:16

packages/shell/src/router/router.ts:192

---

shell.Router.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<State<string>>`

Type declaration ▸ (): `Subscribable<State<string>>`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits the current State and every time this *changes*.

`Example`

Subscribe to the *Router*:

```
import { Router } from '@sgrud/shell';
import { from } from 'rxjs';

from(new Router()).subscribe(console.log);
```

Returns `Subscribable<State<string>>`

Callback to a Subscribable.

Defined in packages/shell/src/router/router.ts:212

---

shell.Router.add

add

▸ **add**(`route`): `Router`

Overridden **add** method. Invoking this method while supplying a `route` will **add** the supplied `route` to the *Router* after deleting its child Routes from the *Router*, thereby ensuring that only top-most/root `routes` remain part of the *Router*.

Parameters

Name	Type	Description
`route`	`Route<string>`	Route to **add**.

Returns `Router`

This instance.

Overrides Set.add

Defined in packages/shell/src/router/router.ts:298

---

shell.Router.baseHref

baseHref

• `Readonly` **baseHref**: `string`

Absolute **baseHref** for navigation.

Defined in packages/shell/src/router/router.ts:217

---

shell.Router.bind

bind

▸ **bind**(`this`, `outlet?`, `baseHref?`, `hashBased?`): `void`

Binding helper method. Calling this method will **bind** a handler to the global `onpopstate` event, invoking *navigate* with the appropriate arguments. This method furthermore allows the properties *baseHref*, *hashBased* and *outlet* to be overridden. Invoking the **bind** method throws an error if called more than once, without invoking the *unbind* method in between.

`Throws`

ReferenceError.

Parameters

Name	Type	Description
`this`	`Mutable<Router>`	Mutable polymorphic `this`.
`outlet`	`Element | DocumentFragment`	Rendering outlet for navigated Routes.
`baseHref`	`string`	Absolute baseHref for navigation.
`hashBased`	`boolean`	Wether to employ hashBased routing.

Returns `void`

Defined in packages/shell/src/router/router.ts:324

---

shell.Router.constructor

constructor

• **new Router**()

Singleton *Router* class **constructor**. This **constructor** is called once by the Target decorator and sets initial values on the instance. All subsequent calls will return the previously constructed Singleton instance of this class.

Overrides Set<Route>.constructor

Defined in packages/shell/src/router/router.ts:268

---

shell.Router.handle

handle

▸ **handle**(`state`, `replace?`): `Observable<State<string>>`

Implementation of the **handle** method as required by the Task interface contract. This method is called internally by the *match* method after all RouterTasks have been invoked. It is therefore considered the default or fallback RouterTask and handles the rendering of the supplied `state`.

Parameters

Name	Type	Default value	Description
`state`	`State<string>`	`undefined`	*Router* State to handle.
`replace`	`boolean`	`false`	Wether to replace the State.

Returns `Observable<State<string>>`

Observable of the handled State.

Implementation of Task.handle

Defined in packages/shell/src/router/router.ts:365

---

shell.Router.hashBased

hashBased

• `Readonly` **hashBased**: `boolean`

Wether to employ **hashBased** routing.

Defined in packages/shell/src/router/router.ts:222

---

shell.Router.join

join

▸ **join**(`segment`): `string`

Segment **join**ing helper. The supplied `segment` is converted to a string by *spool*ing to its top-most parent and iterating through all children while concatenating every encountered path. If said path is an (optional) parameter, this portion of the returned string is replaced by the respective Params value.

Parameters

Name	Type	Description
`segment`	`Segment<string>`	Segment to be **join**ed.

Returns `string`

Joined Segment as string.

Defined in packages/shell/src/router/router.ts:413

---

shell.Router.match

match

▸ **match**(`path`, `routes?`): `undefined` | `Segment<string>`

Main *Router* **match**ing method. Calling this method while supplying a `path` and optionally an array of `routes` will return a **match**ing Segment or undefined, if no match was found. If no `routes` are supplied, routes previously added to the *Router* will be used. The **match** method represents the backbone of the *Router* class, as it, given a list of `routes` and a `path`, will determine wether this path represents a **match** within the list of `routes`, thereby effectively determining navigational integrity.

`Example`

Test if path `'example/route'` **match**es `child` or `route`:

```
import { Router } from '@sgrud/shell';

const path = 'example/route';
const router = new Router();

const child = {
  path: 'route'
};

const route = {
  path: 'example',
  children: [child]
```

```
};

router.match(path, [child]); // false
router.match(path, [route]); // true
```

Parameters

Name	Type	Description
`path`	`string`	Path to **match** against.
`routes`	`Route<string>[]`	Routes to use for **match**ing.

Returns `undefined | Segment<string>`

Matching Segment or undefined.

Defined in packages/shell/src/router/router.ts:475

---

shell.Router.navigate

navigate

▸ **navigate**(`target`, `search?`, `replace?`): `Observable<State<string>>`

Main navigation method. Calling this method while supplying either a path or Segment as navigation `target` (and optional `search` parameters) will normalize the path by trying to *match* a respective Segment or directly use the supplied Segment as next State. This upcoming State is looped through all linked RouterTasks and finally *handle*d by the *Router* itself to render the resulting, possibly intercepted and mutated State.

`Throws`

Observable if URIError.

Parameters

Name	Type	Default value	Description
`target`	`string | Segment<string>`	`undefined`	Path or Segment to **navigate** to.
`search?`	`string`	`undefined`	Optional search parameters.
`replace`	`boolean`	`false`	Wether to replace the State.

Returns `Observable<State<string>>`

Observable of the *Router* State.

Defined in packages/shell/src/router/router.ts:574

---

shell.Router.outlet

outlet

• `Readonly` **outlet**: `Element | DocumentFragment`

Rendering **outlet** for navigated Routes.

Defined in packages/shell/src/router/router.ts:229

---

shell.Router.rebase

rebase

▸ **rebase**(`path`, `prefix?`): `string`

Rebasing helper method. **Rebase**s the supplied `path` against the current *baseHref*, by either prepending the *baseHref* to the supplied `path` or stripping it, depending on the `prefix` argument.

Parameters

Name	Type	Default value	Description
`path`	`string`	`undefined`	Path to **rebase** against the *baseHref*.
`prefix`	`boolean`	`true`	Wether to prepend or strip the *baseHref*.

Returns `string`

Rebased `path`.

Defined in packages/shell/src/router/router.ts:631

---

shell.Router.spool

spool

▸ **spool**(`segment`, `rewind?`): `Segment<string>`

Spooling helper method. Given a `segment` (and wether to `rewind`), the top-most parent (or deepest child) of the graph-link Segment is returned.

Parameters

Name	Type	Default value	Description
`segment`	`Segment<string>`	`undefined`	Segment to **spool**.
`rewind`	`boolean`	`true`	**Spool** direction.

Returns `Segment<string>`

Spooled Segment.

Defined in packages/shell/src/router/router.ts:658

---

shell.Router.state

state

• `get` **state**(): `State<string>`

Getter mirroring the current value of the *changes* BehaviorSubject.

Returns `State<string>`

Defined in packages/shell/src/router/router.ts:255

---

shell.Router.unbind

unbind

▸ **unbind**(this): void

Unbinding helper method. Calling this method (after calling *bind*) will **unbind** the previously bound handler from the global onpopstate event. Further, the arguments passed to *bind* are revoked, meaning the default values of the properties *baseHref*, *hashBased* and *outlet* are restored. Calling this method without previously *bind*ing the *Router* will throw an error.

Throws

ReferenceError.

Parameters

Name	Type	Description
this	Mutable<Router>	Mutable polymorphic this.

Returns void

Defined in packages/shell/src/router/router.ts:686

---

shell.Router.changes

changes

• Private Readonly **changes**: BehaviorSubject<State<string>>

Internally used BehaviorSubject containing and emitting every navigated State.

Defined in packages/shell/src/router/router.ts:238

---

shell.Router

Router

• **Router**: Object

Namespace containing types and interfaces used and intended to be used in conjunction with the Singleton Router class.

See

Router

Defined in packages/shell/src/router/router.ts:16

packages/shell/src/router/router.ts:192

---

shell.Router.Left

Left

T **Left**<S>: S extends `${infer I}/${string}` ? I : S

String literal helper type. Represents the **left** part of a path.

Example

Left of 'nested/route/path':

```
import type { Router } from '@sgrud/shell';

const left: Router.Left<'nested/route/path'>; // 'nested'
```

Type parameters

Name	Type	Description
`S`	extends `string`	Route path string type.

Defined in

packages/shell/src/router/router.ts:31

---

shell.Router.Params

Params

T **Params**<`S`>: `S` extends `${string}:${infer P}` ? `P` extends `${Left<P>}${infer R}` ? `Params<R>` : `never` & `Left<P>` extends `${infer O}?` ? { [K in O]?: string } : { [K in Left<P>]: string } : {}

Type helper representing the (optional) **Params** of a Route path. By extracting string literals starting with a colon (and optionally ending on a question mark), a union type of a key/value pair for each parameter is created.

`Example`

Extract parameters from `'item/:id/field/:name?'`:

```ts
import type { Router } from '@sgrud/shell';

const params: Router.Params<'item/:id/field/:name?'>;
// { id: string; name?: string; }
```

Type parameters

Name	Description
`S`	Route path string type.

Defined in

packages/shell/src/router/router.ts:52

---

shell.Router.Segment

Segment

• **Segment**<`S`>: `Object`

Interface describing the shape of a Router **Segment**. A **Segment** represents a navigated Route and its corresponding Params. As Routes are represented in a tree-like structure and one **Segment** represents one layer within the Route-tree, each **Segment** may have a *parent* and/or a *child*. The resulting graph of **Segment**s represents the navigated path through the underlying Route-tree.

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	Route path string type.

Defined in

packages/shell/src/router/router.ts:76

Router.Segment.child

child

• `Optional` `Readonly` **child**: `Segment<string>`

Optional **child** of this *Segment*.

Defined in packages/shell/src/router/router.ts:81

Router.Segment.params

params

• `Readonly` **params**: `Params<S>`

Route path Params and corresponding values.

Defined in packages/shell/src/router/router.ts:89

Router.Segment.parent

parent

• `Optional` `Readonly` **parent**: `Segment<string>`

Optional **parent** of this *Segment*.

Defined in packages/shell/src/router/router.ts:94

Router.Segment.route

route

• `Readonly` **route**: `Route<S>`

Route associated to this *Segment*.

Defined in packages/shell/src/router/router.ts:101

---

shell.Router.State

State

• **State**<`S`>: `Object`

Interface describing the shape of a Router **State**. Router **State**s correspond to the browser history, as each navigation results in a new **State** being created. Each navigated **State** is represented by its absolute *path*, a Segment as entrypoint to the graph-like representation of the navigated path through the route-tree and *search* parameters.

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	Route path string type.

Defined in packages/shell/src/router/router.ts:118

Router.State.path

path

• `Readonly` **path**: `S`

Absolute **path** of the Router *State*.

Defined in packages/shell/src/router/router.ts:125

Router.State.search

search

• `Readonly` **search**: `string`

Search parameters of the Router *State*.

Defined in packages/shell/src/router/router.ts:132

Router.State.segment

segment

• `Readonly` **segment**: `Segment<S>`

Segment of the Router *State*.

Defined in packages/shell/src/router/router.ts:140

---

shell.Router.Task

Task

• **Task**: `Object`

Interface describing the shape of a RouterTask. These **Task**s are run whenever a navigation is triggered and may intercept and mutate the next State or completely block or redirect a navigation.

See

RouterTask

Defined in packages/shell/src/router/router.ts:154

Router.Task.handle

handle

▸ **handle**(`next`): `Observable<State<string>>`

Method called when a navigation was triggered.

Parameters

Name	Type	Description
`next`	`State<string>`	Next State to be handled.

Returns `Observable<State<string>>`

Observable of handled State.

Defined in packages/shell/src/router/router.ts:165

---

shell.RouterLink

RouterLink

• **RouterLink**: `Object`

Custom element extending the HTMLAnchorElement. This element provides a declarative way to invoke the Router, while maintaining compatibility with SSR/SEO aspects of SPAs. This is achieved by rewriting absolute *href*s to be contained within the applications base href and intercepting the default browser behavior when *onclick*ed.

`Example`

A `router-link`:

```
<a href="/example" is="router-link">Example</a>
```

`See`

Router

Defined in

packages/shell/src/router/link.ts:36

---

shell.RouterLink.observedAttributes

observedAttributes

■ `Static` `Readonly` **observedAttributes**: `string`[]

Array of attribute names, which should be observed for changes, which will trigger the *attributeChanged-Callback*. This element only observes the `href` attribute.

Defined in

packages/shell/src/router/link.ts:43

---

shell.RouterLink.attributeChangedCallback

attributeChangedCallback

▸ **attributeChangedCallback**(`_name`, `_prev?`, `next?`): `void`

This method id called whenever the element's `href` attribute is added, removed or changed. The `next` attribute value is used to determine wether to rewrite the `href` by letting the Router *rebase* it.

Parameters

Name	Type	Description
`_name`	`string`	Attribute name (ignored).
`_prev?`	`string`	Previous value (ignored).
`next?`	`string`	Next value.

Returns

`void`

Defined in

packages/shell/src/router/link.ts:83

---

shell.RouterLink.constructor

constructor

• **new RouterLink**()

Public **constructor** of this custom element. This **constructor** is called whenever an instance this custom element is rendered.

Overrides

HTMLAnchorElement.constructor

Defined in packages/shell/src/router/link.ts:62

---

shell.RouterLink.onclick

onclick

• **onclick**: `(event: MouseEvent) => void`

Type declaration ▸ `(event): void`

Overridden **onclick** handler, preventing the default browser behavior and letting the Router handle the navigation instead.

Parameters

Name	Type	Description
`event`	`MouseEvent`	Mouse click event.

Returns `void`

Overrides HTMLAnchorElement.onclick

Defined in packages/shell/src/router/link.ts:102

---

shell.RouterLink.router

router

• `Private Readonly` **router**: `Router`

Factored-in **router** property retrieving the linked Router.

`Decorator`

Factor

Defined in packages/shell/src/router/link.ts:56

---

shell.RouterOutlet

RouterOutlet

• **RouterOutlet**: `Object`

Custom element extending the HTMLSlotElement. When this element is constructed, it supplies the value of its *baseHref* attribute and the presence of a *hashBased* attribute on itself to the Router while *bind*ing the Router to itself. This element should only be used once, as it will be used by the Router as *outlet* to render the current State.

`Example`

A `router-outlet`:

```
<slot baseHref="/example" is="router-outlet">Loading...</slot>
```

`See`

Router

Defined in packages/shell/src/router/outlet.ts:38

---

shell.RouterOutlet.baseHref

baseHref

• `get` **baseHref**(): `undefined` | `string`

Getter mirroring the **baseHref** attribute of the element.

Returns `undefined` | `string`

Defined in packages/shell/src/router/outlet.ts:43

---

shell.RouterOutlet.constructor

constructor

• **new RouterOutlet**()

Custom element **constructor**. Supplies the value of its *baseHref* attribute and the presence of a *hash-Based* attribute on itself to the Router while *bind*ing the Router to itself. It furthermore invokes a `setTimeout` loop, running until the number of routes the router contains evaluates truthy, which in turn triggers an initial navigation.

Overrides HTMLSlotElement.constructor

Defined in packages/shell/src/router/outlet.ts:63

---

shell.RouterOutlet.hashBased

hashBased

• `get` **hashBased**(): `boolean`

Getter mirroring the presence of a **hashBased** attribute on the element.

Returns `boolean`

Defined in packages/shell/src/router/outlet.ts:50

---

shell.RouterTask

RouterTask

• `Abstract` **RouterTask**: `Object`

Abstract base class to implement **RouterTask**s. By Targeting or otherwise providing an implementation of this abstract **RouterTask** base class to the Linker, the implemented *handle* method is called whenever a new State is triggered by navigating. This interceptor-like pattern makes complex routing strategies like asynchronous module-retrieval and the similar tasks easy to be implemented.

`Decorator`

Provide

`Example`

Simple **RouterTask** stub:

```
import type { Router, RouterTask } from '@sgrud/shell';
import type { Observable } from 'rxjs';
import { Provider, Target } from '@sgrud/core';

@Target<typeof ExampleRouterTask>()
export class ExampleRouterTask
  extends Provider<typeof RouterTask>('sgrud.shell.router.RouterTask') {

  public override handle(
    prev: Router.State,
```

```
    next: Router.State,
    handler: Router.Task
  ): Observable<Router.State> {
    throw new Error('Stub!');
  }

}
```

See

- Route
- Router

Defined in packages/shell/src/router/task.ts:48

---

shell.RouterTask.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.router.RouterTask"`

Magic string by which this class is provided.

See

provide

Defined in packages/shell/src/router/task.ts:57

---

shell.RouterTask.constructor

constructor

• **new RouterTask**()

---

shell.RouterTask.handle

handle

▸ `Abstract` **handle**(`prev`, `next`, `handler`): `Observable<State<string>>`

Abstract **handle** method, called whenever a new State should be navigated to. This method provides the possibility to intercept these upcoming States and, e.g., mutate or redirect them.

Parameters

Name	Type	Description
`prev`	`State<string>`	Previously active router state.
`next`	`State<string>`	Next router state to be activated.
`handler`	`Task`	Next task handler.

Returns `Observable<State<string>>`

Next handled router state.

Defined in packages/shell/src/router/task.ts:72

---

shell.component

component

• `Const` **component**: typeof `component`

Unique symbol used as property key by the Component decorator to associate the supplied constructor with its wrapper.

Defined in packages/shell/src/component/component.ts:10

---

shell.createElement

createElement

▸ **createElement**(`type`, `props?`, `ref?`): `Element`

JSX element factory. Provides JSX runtime-compliant bindings to the incremental-dom library. This factory function is meant to be implicitly imported by the transpiler and returns an array of bound incremental-dom function calls, representing the created JSX element. This array of bound functions can be rendered into an element attached to the DOM through the render function.

See

render

Parameters

Name	Type	Description
`type`	`Function` | keyof `HTMLElementTagNameMap`	Element type.
`props?`	`Record<string, any>`	Element properties.
`ref?`	`Key`	Element reference.

Returns `Element`

Array of bound calls.

Defined in packages/shell/src/component/runtime.ts:114

---

shell.createFragment

createFragment

▸ **createFragment**(`props?`): `Element`

JSX fragment factory. Provides a JSX runtime-compliant helper function used by the transpiler to create JSX fragments.

Parameters

Name	Type	Description
`props?`	`Record<string, any>`	Fragment properties.

Returns `Element`

Array of bound calls.

Defined in packages/shell/src/component/runtime.ts:179

---

shell.customElements

customElements

• `Const` **customElements**: `CustomElementRegistry` & { `getName`: (`constructor`: `CustomElementConstructor`) => `undefined` | `string` }

Proxy around the built-in customElements object, maintaining a mapping of all registered elements and their corresponding names, which can be queried by calling *getName*.

Remarks

https://github.com/WICG/webcomponents/issues/566

Defined in

packages/shell/src/component/registry.ts:15

---

shell.references

references

▸ **references**(`target`): `Map<Key, Node>` | `undefined`

JSX **references** helper. Calling this function while supplying a viable `target` will return all referenced JSX elements mapped by their corresponding Keys known to the supplied `target`. A viable `target` may be any element, which previously was `target` to the render function.

Parameters

Name	Type	Description
`target`	`Element`	`DocumentFragment`	Element to lookup **references** for.

Returns

`Map<Key, Node>` | `undefined`

Resolved **references**.

Defined in

packages/shell/src/component/runtime.ts:207

---

shell.render

render

▸ **render**(`target`, `element`): `Node`

JSX **render**ing helper. This helper is a wrapper around the *patch* function from the incremental-dom library and **render**s a JSX `element` created through createElement into an `target` element or fragment.

See

createElement

Parameters

Name	Type	Description
`target`	`Element`	`DocumentFragment`	Element or fragment to **render** into.
`element`	`Element`	JSX element to be **render**ed.

Returns

`Node`

Rendered `target` element.

Defined in

packages/shell/src/component/runtime.ts:229

---

shell.route

route

• `Const` **route**: typeof `route`

Unique symbol used as property key by the Route decorator to associate the supplied route configuration to the decorated element.

Defined in

packages/shell/src/router/route.ts:64

Module: state

state

• **state**: `Object`
